Simulating Human-Robot Collaboration with Virtual and Augmented Reality

Undergraduate Honors Thesis

Presented in Partial Fulfillment of the Requirements for Graduation with Distinction in the Department of Mechanical Engineering at The Ohio State University

By Roger Kassouf

November 2018

Advisor: Haijun Su, Ph.D.

Abstract

Robots do work for humans that is often undesirable, whether dirty, dull, or dangerous. As the pace of automation continues, division and sharing of labor between man and machine is expected to increase in manufacturing. Conventional robots exert high forces and are difficult to program, limiting their use to large-scale enterprises. Collaborative robots ("cobots") are an alternative that resolve these issues, due to built-in force-limiting sensors and intuitive hands-on programming. This allows a human to perform complex and delicate tasks, while the robot performs undesirable tasks.

However, small-to-medium scale enterprises do not fully know what possibilities exist for including a cobot in their operations without physically obtaining a robot. Nonphysical design tools for robotic systems are built on software that is targeted to a technical audience.

The research aimed to develop an interactive design tool for the implementation of cobot collaboration, that would be accessible to a non-technical audience. This was done leveraging virtual reality (VR) and augmented reality (AR) technology to create an immersive experience with digital content. While VR is limited to a purely digital experience, AR allows users to superimpose VR elements in the real world. The Unity 3D game engine was used as the software development platform, paired with the Microsoft Hololens VR/AR headset as the human-simulation interface. A Universal Robots UR-10 cobot manipulator was modelled to accurately reflect its size, shape, and movement behavior. A virtualized scenario of metal grinding and polishing was then created with the UR-10 cobot model acting as an assistant to the human.

The design tool is considered successful at meeting the objectives for simulating human-robot collaboration. It may open opportunities for non-technical persons in manufacturing to understand how to implement a cobot in their enterprise. Furthermore, the tool may serve as a basis for future developments into advanced manufacturing design with VR/AR and collaborative robotics.

Acknowledgements

I am very thankful for the support of my research advisor, Dr. Haijun Su. Throughout the research, he has been there both as a guiding and challenging presence. I am ever grateful for this incredibly unique research opportunity, which was made possible in no small part through him.

I have great appreciation for Tyler Morrison, who has also advised me throughout the project. His moderating presence and out-of-the-box thinking often gave me the fresh perspective I needed whenever I was stuck.

A special thanks to Cameron Spicer, who was my research partner during his own Undergraduate Honors Thesis. His advancements set forth in his thesis were fundamental in my own. I look back fondly on our collaboration and his company.

I am grateful to have learned a great deal from Dr. Robert Siston during my two semesters in MECHENG 4999H. His mentorship has encouraged to ask tough questions, to give clear answers, and to strive towards professionalism in engineering.

I would like to thank some past members of our lab that have contributed, big or small, to this thesis. Jing Huang and Hanyuan Xu were the first to start the project. They cleared a lot of headway in unknown territory for Cameron and me to be more productive in our own research. Anil Turkkan was very helpful and knowledgeable whenever I had a question. I wish them all the best in their endeavors.

Last and especially not least, I am so grateful to be blessed with my ever-supportive family and friends. They have really helped my research by listening to my presentations and going through my proposal. Most importantly, they have been there for me when I needed them the most. I cannot thank them enough.

Table of Contents

List of Figures

List of Tables

Chapter 1: Introduction

Robots have been used for more than 50 years to assist humans in tasks which are mundane, repetitive, or dangerous [1]. During the start of the "Digital Revolution" (Industry 3.0), robots began to emerge from a leap in "digital computing and communication technology, enhancing systems' intelligence" [2]. Most recently, there is a push to call recent advancements in technology the "Information Revolution", or "Industry 4.0" - where highly complex information networks can facilitate the creation of artificial intelligence, big data analytics, and more sophisticated systems.

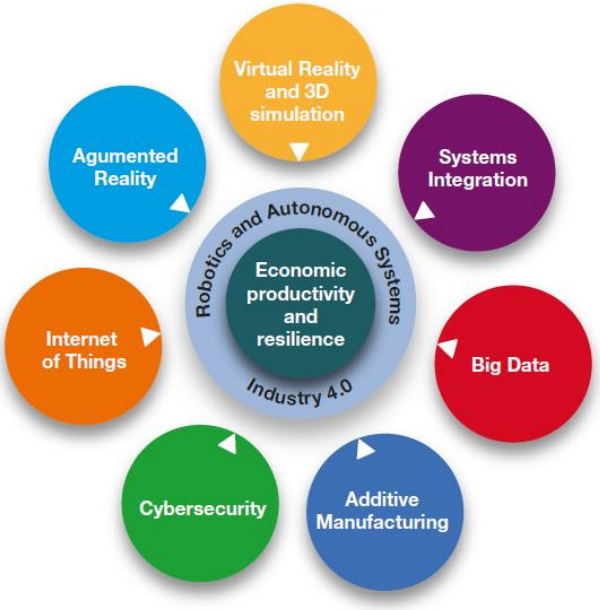


Figure 1 UK Robotics and Automation Systems, needs for Industry 4.0 [2]

Today, it is difficult to imagine a world without robots, especially in manufacturing. There are about 66 industrial robots for every 10,000 employees in manufacturing worldwide, but in countries like in Japan, Germany and the United States, we find 1.5, 1.2 and 1.1 thousand robots per every 10,000 workers, respectively [3]. Experts anticipate even more jobs will be assisted by robots or become automated in the future, reaching across more markets than just manufacturing. Some of these markets include medicine, logistics, transportation, and other areas requiring skilled labor. As the proliferation of automation continues across multiple industries, the way in which humans and robots interact is critical for the successful integration of robots into the workplace.

Figure 2 Use of conventional robots in Tesla's highly-automated production line [4]

Despite the widespread utilization of robots, businesses still face significant hurdles to adopt them more fully into their operations. Conventional robots have had a well-defined role in business - they are programmed to fulfill a specific set of tasks repetitiously. However, programming for these tasks is usually done by highly qualified individuals, such as technicians and engineers, rather than lay persons. It is prohibitively difficult for non-technical persons to understand how robots work at a functional level, due to the complexity of the models involved. In addition, another barrier to working with conventional robots is that they have a limited amount of contextual understanding about their environment. For safety reasons, humans are not allowed to enter the workspace of the robot during its operation. Large manufacturing corporations with significant capital may be able to overcome these barriers by investing in highly-skilled workers and the infrastructure to accommodate the robots. Small businesses and non-manufacturing markets, however, lack the resources to efficiently surmount these challenges. There is obviously, then, a great need for robots to be more flexible to better satisfy the requirements of many industries.

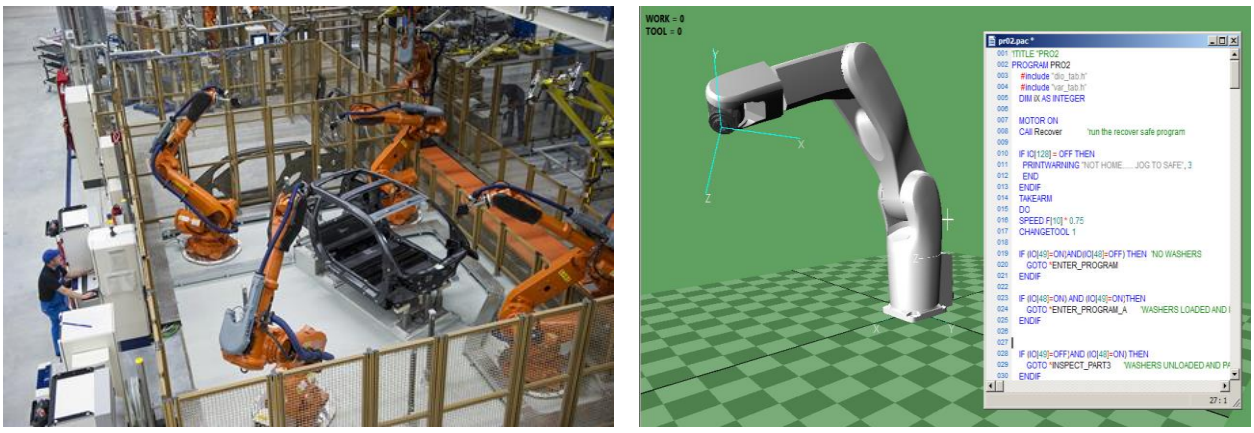


Figure 3 (Left) Manufacturing associate separated from guarded robots [5]
(Right) Text-based programming of a robot. [6]

An available solution on the market for manufacturing is a "collaborative robot", or "cobot". Cobots are human-safe by removing pinch points, and mitigating impact through force and torque-limiting technology. The human operator can then program the cobot through dragging it through

waypoint positions, and logging commands through a companion tablet. While a cobot manipulator is more expensive than a conventional variant, decreases in safety and programming cost for an example case could save approximately $75k USD [7], described in more detail in Table 1. Examples of cobots are the Universal Robots series (UR-3, UR-5, and UR-10) and Rethink Robotics' Baxter and Sawyer.

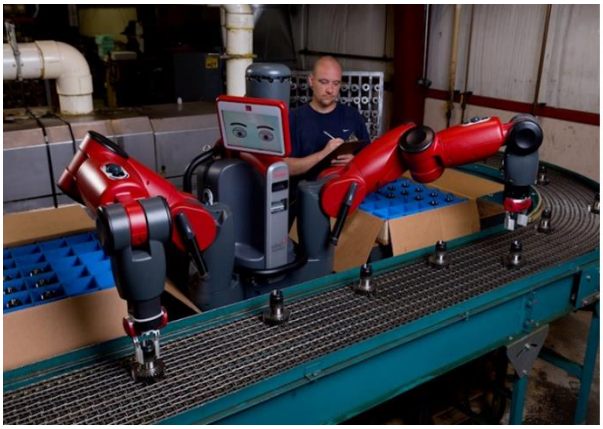



Figure 4 (Left) Baxter used in a palletizing operation [8]
(Right) Associate programs a UR cobot manipulator for CNC mill tending [9]

Table 1 Proposed budgets by comparing similar collaborative and conventional robots

Item	Collaborative	Conventional
Robot model	UR-5	Fanuc LR Mate 200iA
Base Price	$35,000	$25,000
Tooling & Accessories	$7,000-$15,000	$7,000-$30,000
Safety & PLCs	$0	$7,000
Integration Time	$0	$10,000
Programming	$0	$(Varies)
BUDGET	**$50,000**	**$125,000**

Many groups are pursuing research in human-robot collaboration. At John Hopkins University's Laboratory for Computational Sensing and Robotics, researchers leveraged the Oculus Rift Virtual Reality headset and environment virtualization to control an industrial robot manipulator remotely (see Figure 3) [10]. ProFactor, an Austrian group, performed a human-robot collaborative

assembly process using vision systems to model the human and environment, and a Universal Robots UR10 robot manipulator with a gripper tool [11]. Continuing research and development is critical to the advent of collaborative robotics across all markets and businesses around the world.

Virtual reality (VR) and augmented reality (AR) technologies, along with 3D simulation, are another part of the UK RAS projection for key components of Industry 4.0 [2]. Modelling of 3D objects is standard practice in many fields of engineering and science. However, conventional visualization of these models has been limited to a 2D view on a screen. Stereoscopic projections of 3D objects provide a sense of visual perspective that matches well with human binocular vision. VR can use stereoscopic projections to create an immersive experience with digital objects for the user. The limitation of VR is that while the user is fully immersed in the virtual world, they are unable to see what is happening in the real world.

Augmented reality instead takes these digital elements and superimposes them onto the real world. The Microsoft Hololens is an AR device that is commercially available and mobile. It creates "holograms", which when superimposed can stay in position in the real world. This allows the user to naturally view the 3D object by looking at it and walking around it. In addition, the user can interact with VR elements on the Hololens through the user's Gaze (where they are looking) and Gestures (motions tracked by the user's hand). Description of each may be found in Appendix C.

Figure 5 A Hololens user views a 3D model of a machine superimposed on the table [12]

Similar efforts to those presented in this thesis have been performed at the Design Innovation and Simulation Laboratory in the Department of Mechanical and Aerospace Engineering at The Ohio State University, through the same project. This project leverages the Unity game engine to perform simulations capable of being exported to the Hololens as an app. For readers unfamiliar with Unity, it is highly recommended to read Appendix A: Introduction to Unity.

Jing Huang and Hanyuan Xu started the project over Summer 2017 and were able to perform a simulated by-hand example of a toy car, albeit without a fully-functioning robot. Cameron Spicer completed a similar thesis to mine with the same general objectives, and I assisted him with the project during that time [13]. Through his thesis, Cameron was able to complete a solid and kinematic model of the cobot, could drag the cobot's end effector with Gesture control, and modelled a sample manufacturing scenario. His work will be referenced at times in the thesis.

1.1 Focus of Thesis

The research aimed to develop an interactive design tool for the implementation of cobot collaboration, that would be accessible to a non-technical audience. The design of the tool was guided by these objectives:

1. Present a physically and operationally accurate model of an available cobot.
2. Design a realistic manufacturing scenario which could be a good candidate for human-robot collaboration.
3. Provide an example interactive experience for a human with the cobot and manufacturing scenario.

My contribution is adding more functionality to the tool than was done in [13]. The cobot can now move autonomously rather than only being dragged by the human. I have modelled the Teach Pendant of the UR-10 to give it motion instructions and to program it. Finally, I have added a new manufacturing scenario. Throughout, there are improvements on the robustness and complexity of the tool. The tool is developed in Unity and is deployed to the Microsoft Hololens.

1.2 Significance of Research

The proposed research tackles a subset of the problems presented through improving understanding of human-robot collaboration, while maintaining relevance to the larger challenge of successfully implementing robotic and automated solutions in the global economy.

With such a tool, small-to-medium size enterprises could be more effective at introducing robotics and automation into their business operations. Instead of going through the time, effort, and money of physical hardware, a user of the tool could explore new uses for human-robot collaboration. This would increase their time-to-market and bolster their competitiveness.

Lastly, the research presents methods that can be used more generally to accelerate related advanced manufacturing research.

1.3 Overview of Thesis

The thesis is split up into chapters which follow the progression of the project from the background knowledge to the conclusions. As you have already read, Chapter 1 focused on introducing the pertinent background, motivation, purpose, and objectives. Chapters 2 details how the UR-10 cobot was developed, from the basic solid and kinematic modelling to more advanced features like motion and “teach” programming. Chapter 3 looks at the development of the manufacturing scenario, from concept generation to finished product. Both Chapters 2 and 3 cover the design results and methodology following all three objectives set out by the research. Chapter 4 is a commentary on the solution, testing it and running through the scenario. Lastly, Chapter 5 states determines contributions made per the purpose and objectives, makes conclusions, and sets out opportunities for future work.

Chapter 2: Development of the Cobot and Teach Pendant

The motivations for using the UR-10 robot as the cobot of choice for our preliminary design tool were presented in [13]. The key metrics of the cobot are listed below:

Table 2 Key metrics of the UR-10 cobot [14]

Degrees of freedom (DOF)	6 rotating joints
Maximum payload	10 kg
Reach	1300 mm
Speed of tool, typical	1 m/s
Repeatability	$\pm$ 0.1 mm
Footprint	∅ 190 mm
Materials	Aluminum, ABS plastic, PP plastic

The metrics here support the choice of the UR-10. Having 6 DOF gives the robot flexibility in precisely locating and rotating a work piece. The payload of 10 kg is usually more than sufficient for supporting end effectors or whatever the robot can grab. Having the 1300 mm reach gives it a great range to comfortably work. Good speed and repeatability mean that it can work quickly and precisely. A 190 mm footprint is rather small, which allows for flexibility in placing the cobot. Lightweight material choices would help with control in real life, due to lower inertia.

A picture of the cobot is presented in the figure below, with its links annotated. Since the robot has 6 joints, it must have 7 links. The first link is the base, which is the connection back to ground. The remainder of the links in the chain are somewhat named after the human arm: Shoulder, Upper Arm, Lower Arm, Wrist 1, and Wrist 2. The last link is the tool flange, which would allow an end effector to attach to it. An illustration is presented in the figure below.

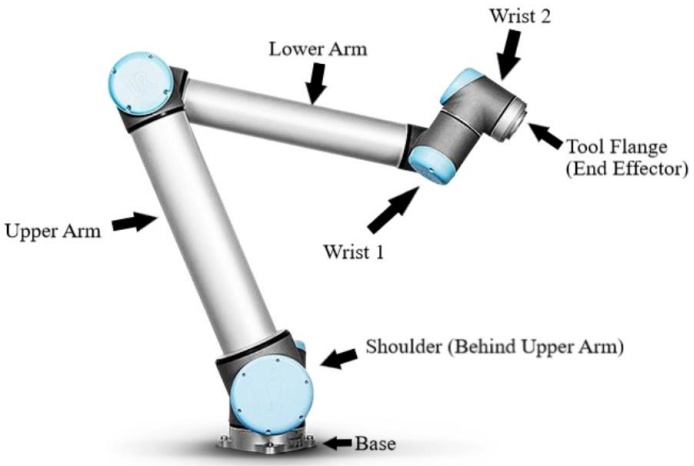


Figure 6 Anatomy of the UR-10 robot [13]

The remaining paragraphs will present a combination of logical continuations of where previous work left off, and completely original ideas that have not been addressed thus far.

2.1 Solid and Kinematic Modelling

The solid body model of the robot began with STEP files provided by Universal Robots for the UR-10, which gave colorless solid bodies for each of the bulk parts of the robot. These parts were opened in SOLIDWORKS. The assembly of these parts contained more parts than links. In example, the Upper Arm link is composed of three parts: a shoulder hub, upper arm hub, and an aluminum connecting rod. Universal Robots uses identical bulk parts to assemble the UR-10. In example, Wrist 1 and Wrist 2 are from same bulk part, even though the links are kinematically distinct. To consolidate the 11 parts into the 7 links, parts which had no revolute relationship between each other were combined into one lumped part, which represented a single link. These "link" parts were then mated in SOLIDWORKS so that they could rotate with respect to each other. The parts were also colored to match the material color of the UR-10. A snapshot from SOLIDWORKS is presented below.

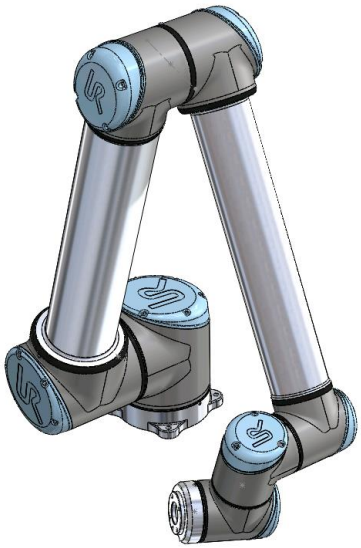


Figure 7 Corrected SOLIDWORKS assembly of UR-10 links

A new process was required to obtain Unity GameObjects with the same size, shape, and color properties as the SOLIDWORKS robot assembly. Previous processes during the time of Huang and Xu's research exported STEP files from SOLIDWORKS, and then used Autodesk 3ds Max to export the files as OBJ files, which can be read by Unity. There were some problems with this method, notably that the color of the parts was stripped, and that the scale was wrong upon export.

The new method was presented in [13] as "Appendix B: Creating Custom Objects in Unity". In my thesis I have isolated the process down to the operational conversions required and is presented in Appendix B of this thesis. 3ds Max can natively import *.sldprt SOLIDWORKS part files, but the issue was getting the appropriate color and scale. The color and scale was corrected by exporting as an *.fbx file, which is an Autodesk format especially designed for video game middleware. It exports the file with the mesh, which is what makes the part's shape appear correctly, along with the constituent materials (read – colors) that come along with the part.

It is appropriate to mention that from this point onward, it is expected that the reader knows at a cursory level what Unity is and how objects are classified in it. For additional background, please refer to Appendix A: Introduction to Unity.

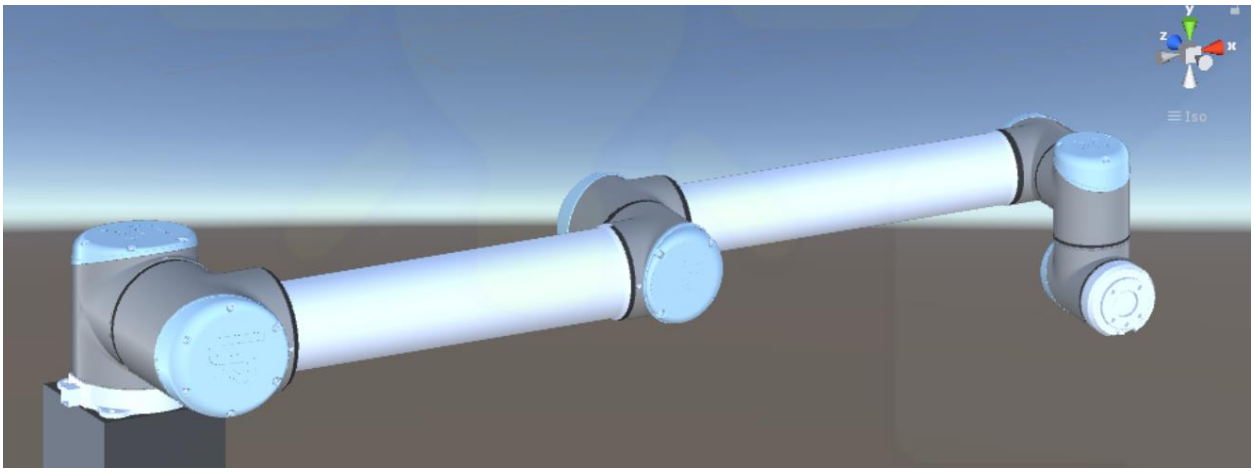


Figure 8 Assembled UR-10 cobot in Unity

With the UR-10 links in Unity, it was time to string them together to form the kinematic chain of the robot. Previous attempts from Huang and Xu's research explored two different ways of creating an acceptable kinematic model of the robot which could be used in Unity. An analytical method was first attempted to determine forward and inverse kinematics. The closed-form equation of the UR-10 were solved in a paper through use of the Denavit–Hartenberg (D-H) convention. The figure below contains a diagram of the figure displaying the corresponding D-H coordinate frames. Notice that the robot is instantiated in Unity following the same convention.

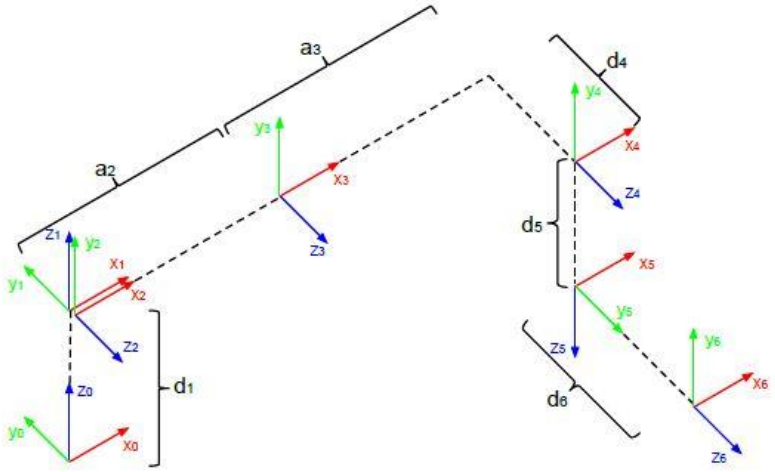


Figure 9 D-H coordinate frames for a UR robot arm [15].

Despite having a closed-form solution available, technical challenges in C# implementation of the algorithms rendered them unusable for the simulation. Instead, Unity's built-in physics engine PhysX contains Hinges, which operate similarly to the Hinge mates in SOLIDWORKS. This enabled the creation of a kinematically-accurate representation for the UR-10 in Unity without

going through the hassle of implementing analytical kinematics. The Hinge Component is presented below, along with the UR-10 with its chain of Hinges.

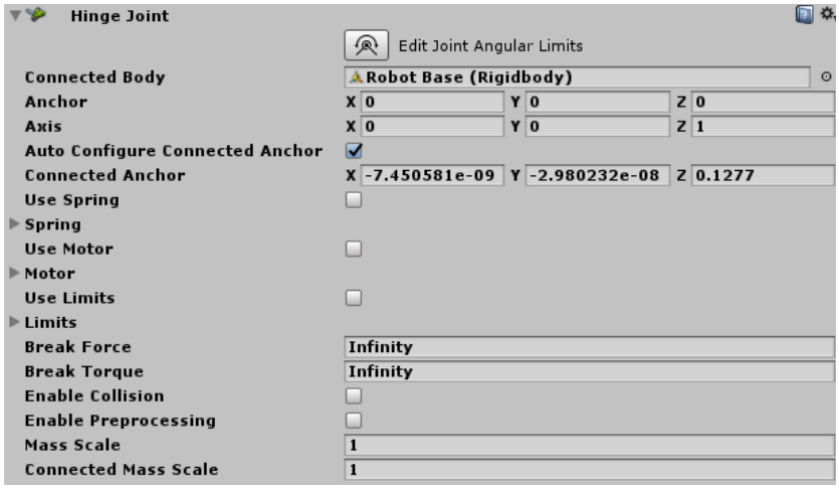


Figure 10 Hinge Component, in this case on the Shoulder GameObject

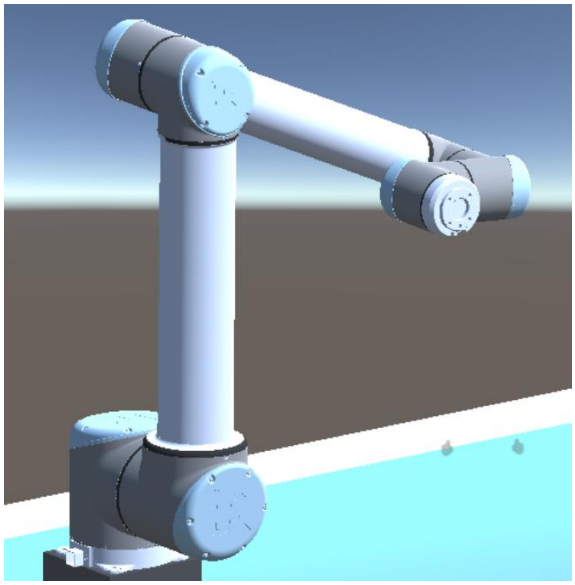


Figure 11 UR-10 cobot, with working Hinge joints

While largely effective, the UR-10 model in [13] had some issues with maintaining physical realism of the robot. Notably during dragging and motion, the robot would tend to oscillate and break apart. The only solution was to set the robot's links' Rigidbody Comopnents to "Is Kinematic", which prevented them from responding to in-game stimuli without additional code. This limited the speed at which the robot could move without self-destruction. Furthermore, the robot could self-intersect and cause undesirable behavior. Before continuing with discussion of the Rigidbody Component, a default Component as viewed in the Inspector is presented again.

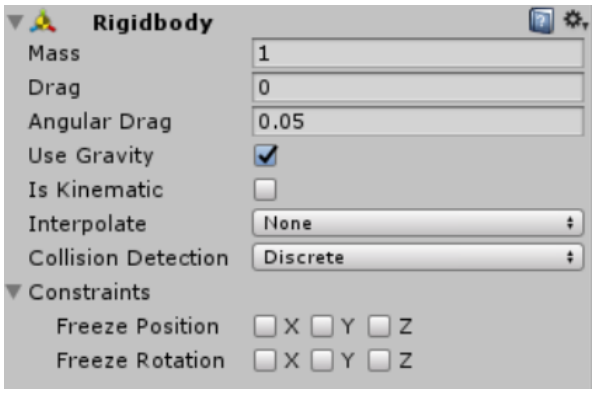


Figure 12 Rigidbody Component with default parameters

For reducing oscillations without setting all the parts to "Is Kinematic", each link object's Rigidbody parameter was modified. It should be noted that unless the robot's links are not Kinematic ("dynamic" as a replacement), then hinge joints do not work. First, only the base was set to Kinematic. This allowed the robot to always solve the kinematics. To reduce the oscillations, the Mass, Drag, and Angular Drag parameters were modified to achieve good results. The table below states the parameters for each Rigidbody component of the link:

Table 3 Rigidbody parameters for UR-10 link GameObjects

Link GameObject	Mass	Drag	Angular Drag
Base	1e+09	0	Infinity
Shoulder	1	0	1
Upper Arm	1	0	1
Lower Arm	1	0	Infinity
Wrist 1	1	0	Infinity
Wrist 2	1	0	Infinity
Tool Flange	1e+09	1	1

While these values were found by trial-and-error observations, there may be some reasonable logic behind them. Unity recommends to not directly change the position of physics objects, but to rather implore a PhysX "force" on them. This would work for most simple bodies, but the robot works upon accurately defined positions. Problematically, directly editing of position changes to the tool flange may induce high internal forces and torques. The increased angular drag "stiffens" the revolute joints, making them behave more normally. Increasing the mass at both the base and the tool flange appears to have a clamping effect on the entire kinematic chain.

The problem of the robot self-intersecting was not entirely corrected but was greatly assisted by assigning the above properties. An attempt was made to modify the upper arm-to-lower arm joint angle limits in the Hinge property. However, this did not truly make a difference and only lead to increased problems. Instead, the Shoulder and Upper Arm Rigidbody components have an Angular Drag value of 1. The joint between these two links leads to less "joint stiffness", thereby allowing the robot to swivel away from itself instead of self-intersecting.

Aside from these changes, improvements to the robot's collision model were made. Unity characterizes a GameObject's collisions through Colliders, which can be of varying shapes: Box, Capsule, Spherical, and Mesh. Multiple colliders of any shape can be attached to the same GameObject for increased fidelity. For all parts except the Tool Flange, Box Colliders were used to good effect. For the Tool Flange, a Mesh Collider was used for increased accuracy. The mesh generated from SOLIDWORKS is the mesh that the Collider is based off, which often creates a high number of mesh vertices that can slow down simulation. This was resolved by converting the mesh to "Convex", and setting the Skin Width down to 0.005. This has an effect of creating a mesh with less, but larger and less precise vertices.

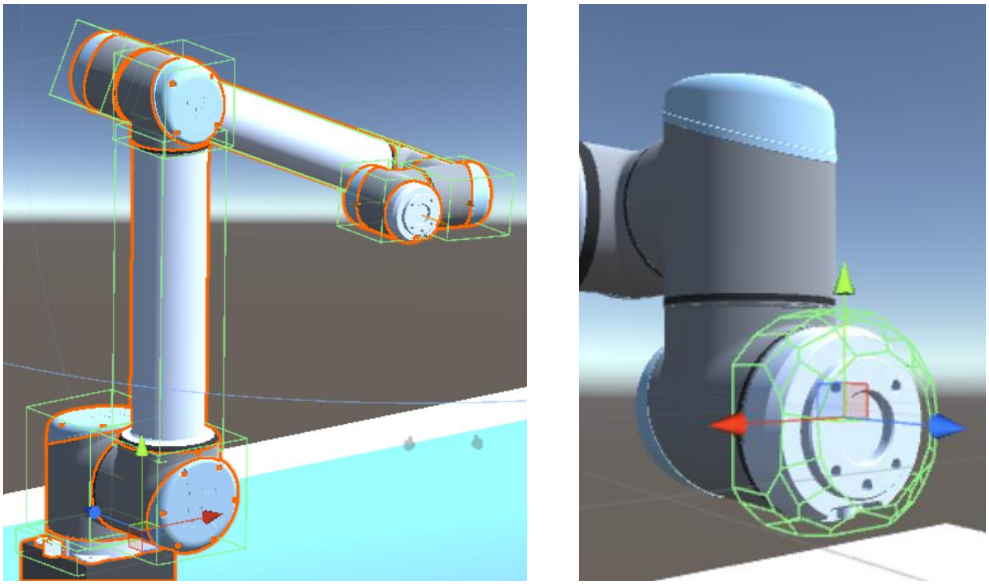


Figure 13 (Left) UR-10 Colliders, except for (Right) Tool Flange Mesh Collider

The UR-10's boundary radius was originally assigned to match the work space limit of 1300 mm, or 1.3 m. If the tool flange reached this limit, however, the robot still had a risk of breaking apart due to kinematic singularities. The boundary radius in the simulation is modified to be slightly smaller at 1.2 m, which shows to be effective at keeping the robot together. When the boundary radius is hit, it now appears until the tool flange returns within the sphere volume.

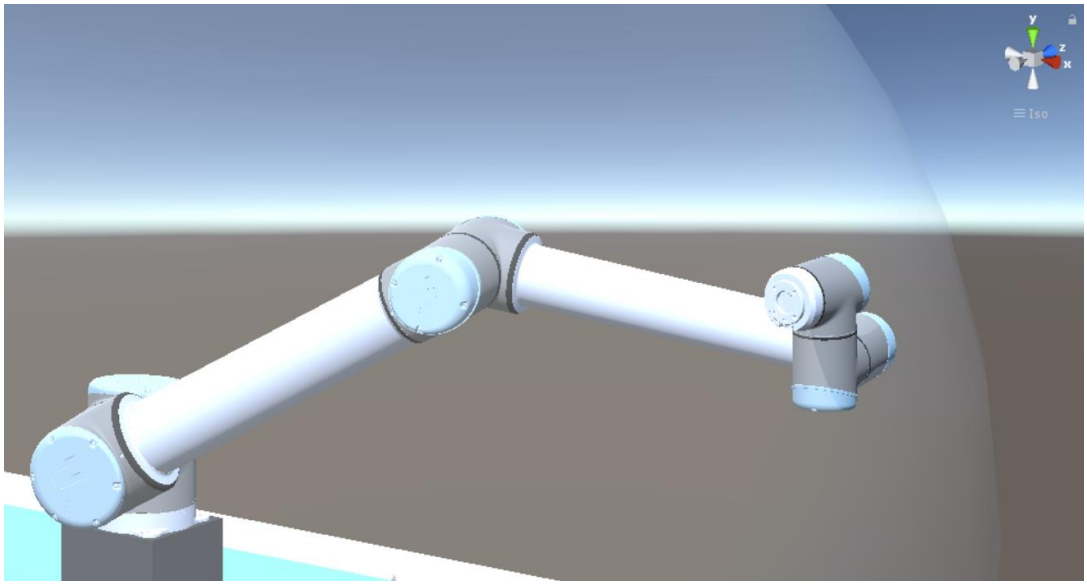


Figure 14 UR-10 reaching its boundary radius and stopping

In summary, improvements were made to present a visually accurate and kinematically stable robot. The more robust robot model is ready to be controlled.

2.2 Motion Behavior Modelling

Originally, the only way that the robot was able to move was through dragging of the Tool Flange to the desired position. Inherently the biggest part of a robot was missing from this – the ability for the robot to move on its own, without the direction of a human. Controlled motion in Unity is not so easily simulated, as the classical method of doing so is to use Rigidbody components with forces. Instead, some control is required so that the end effector (here, the Tool Flange) can be placed exactly where it is desired. The rotation of the tool flange could also be of some importance to the user. The 3d vector position X and quaternion rotation Q of an object defines what is known as a Transform $T = (X, Q)$ in Unity.

The objective of autonomous positioning is to interpolate the Transform of the robot Tool Flange $T_R = (X_R, Q_R)$ to a target or "Waypoint" Transform $T_W = (X_W, Q_W)$. A state machine was conceived to manage autonomous positioning; the script that implements this is RobotMotionStateMachine_V3. The script is a component of the Tool Flange, which is itself a child object of the UR-10 robot, the latter of which contains the rest of the robot links. The figure below lists some of the key parameters as public variables.

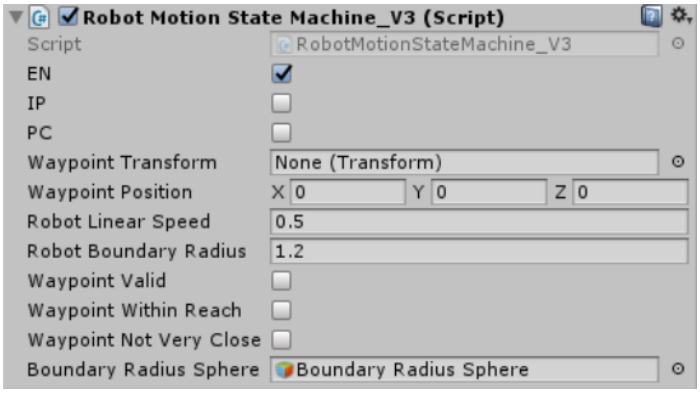


Figure 15 RobotMotionStateMachine_V3 Script Component Public Variables

The Booleans EN, IP, PC are the main control states of the state machine, standing for Enabled, In Progress, and Process Complete, respectively. There is an addition state, WaypointValid, which specifies whether the selected Waypoint is neither too far or too far to the Tool Flange's current position. These are denoted by WaypointWithinReach and WaypointNotVeryClose, respectively. All public variables may be changed either within the script, or external to it.

We desire that the Tool Flange moves at a constant linear speed v, which can be set between 0 and 1 m/s. Let the repeatability distance and boundary sphere radius be $D_1 = 0.001\ m$ and $D_2 = 1.2\ m$, respectively. These three values are pertinent in the process flow for the state machine, which is presented in Figure 16

While the flowchart does a thorough job in describing the minutia of how the state machine works, it may be beneficial to look at things at a high-level first. Most of the operations occur in the FixedUpdate loop of the script, which runs at a refresh rate no greater than 50 Hz. The Waypoint Transform is read in, and a one-cycle delay is used to check if the Waypoint is set at a valid position for the Tool Flange. Assuming it is, the movement is attainable. However, the built-in Unity interpolators need to know estimates of the time and speed of the interpolation. As we have assumed constant speed, these estimates follow simple 1D kinematics: $v = d/t$ or variations of it. With these estimates, we can calculate the interpolated position X_R' using *SmoothDamp* Vector3 interpolation, and the interpolated rotation Q_R' using the *RotateTowards* Quaternion rotation. The true position and rotation of the Tool Flange are then set to the interpolated positions. When the Tool Flange and Waypoint positions are within the repeatability distance D_1 of the UR-10, the movement is assumed to be complete. In the case where the Waypoint position exceeds the boundary sphere of the robot, the robot will freeze wherever it is until the Waypoint returns inside the boundary sphere.

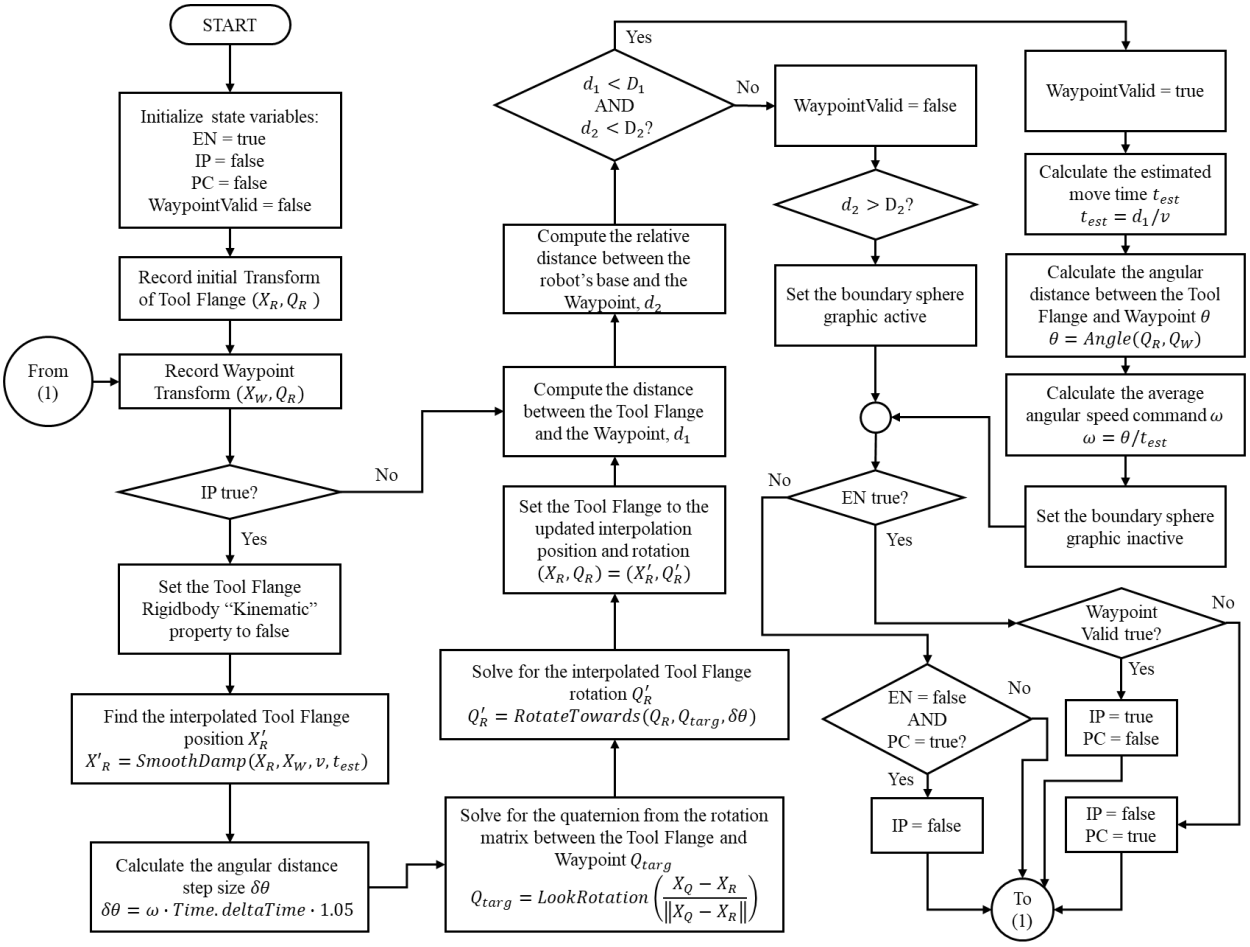


Figure 16 RobotMotionStateMachine_V3 flowchart

To validate that the motion worked, a series of objects to serve as Waypoints were placed in the scene. An arrow object was created to track both position and rotation graphically, as illustrated in Figure 17. The objects were sequentially ordered, and the Waypoint of the state machine updated whenever the waypoint object in the sequence was reached. Figure 18 shows the Tool Flange progressing through these positions.

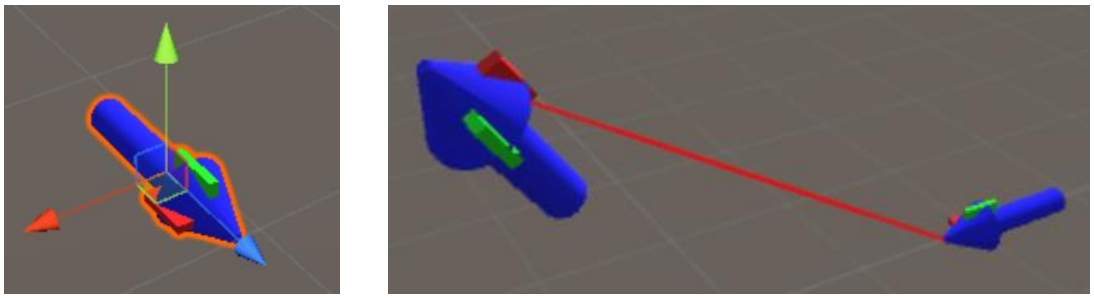


Figure 17 (Left) Arrow created to visualize position and rotation graphically, (Right) Sequence of Waypoint objects for testing motion state machine

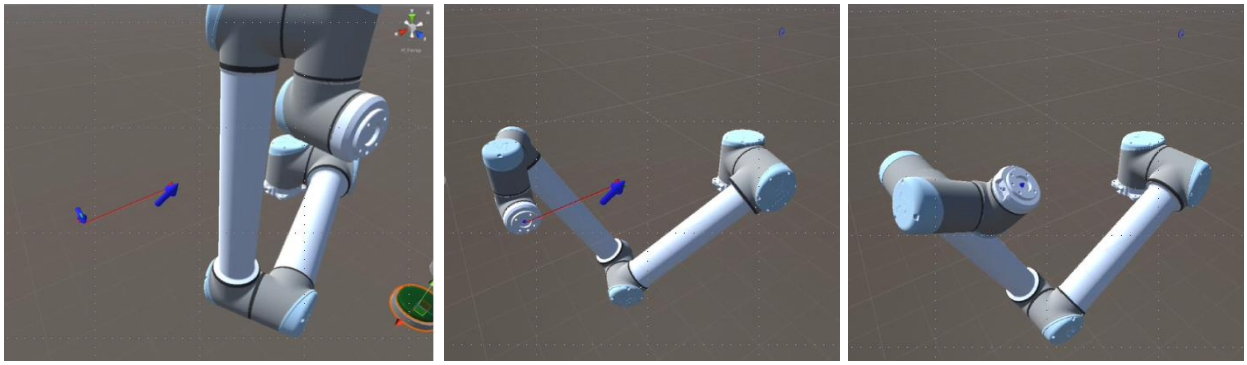


Figure 18 Test of motion state machine reaching preset waypoints: (Left) Before motion is enabled, (Middle) Reaching the first waypoint, and (Right) Reaching the second waypoint

While these arrow objects were convenient for validating motion, they were only useful in programming robot positions within the Unity editor. For the sake of the design tool, it would be unnatural to have the user program key positions with arrows. Instead, they would want to drag the Tool Flange around. Before jumping into how to program these key positions, the ability to have the robot follow the hand with the new state machine enabled was desired.

A new GameObject called “Dragging Waypoint” became the permanent GameObject to serve as the Waypoint for the state machine. With the HandDraggable script as a component of the Dragging Waypoint, the user could guide the robot wherever they wanted through the Tool Flange following the Waypoint.

Some extra features were added to the Dragging Waypoint to help the user understand the behavior of the state machine. A halo light was placed around the Dragging Waypoint when the HandDraggable script was enabled. The halo was color-coded based on if it was actively dragging, and if it the Dragging Waypoint exceeded the boundary sphere of the robot. The waypoint will snap back after releasing the grab so that it re-enters the boundary sphere at the same angle, just closer to the center. Figure 19 displays the effects. The features are implemented in the DraggingWaypointManager script - a flowchart of the script is available in Figure 20.

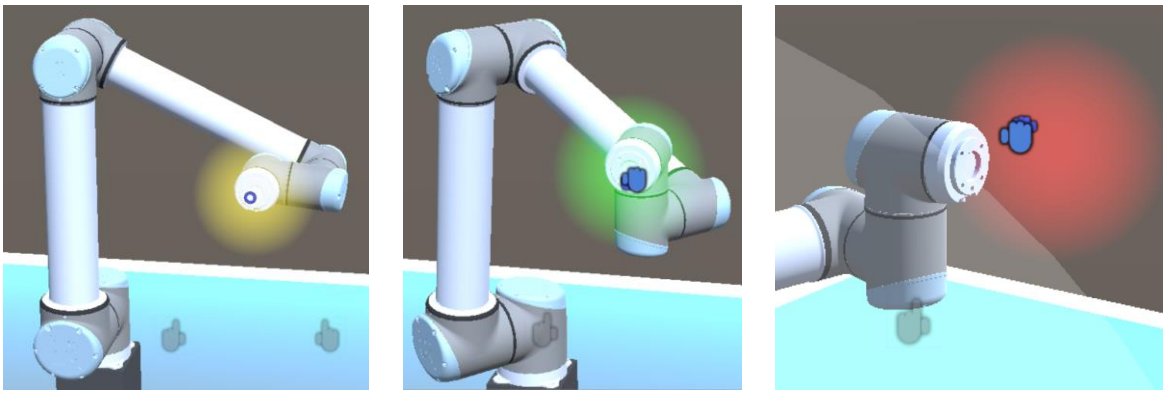


Figure 19 Dragging Waypoint when (left) dragging enabled, but not selected, (center) when within the boundary sphere, and (right) outside the boundary sphere

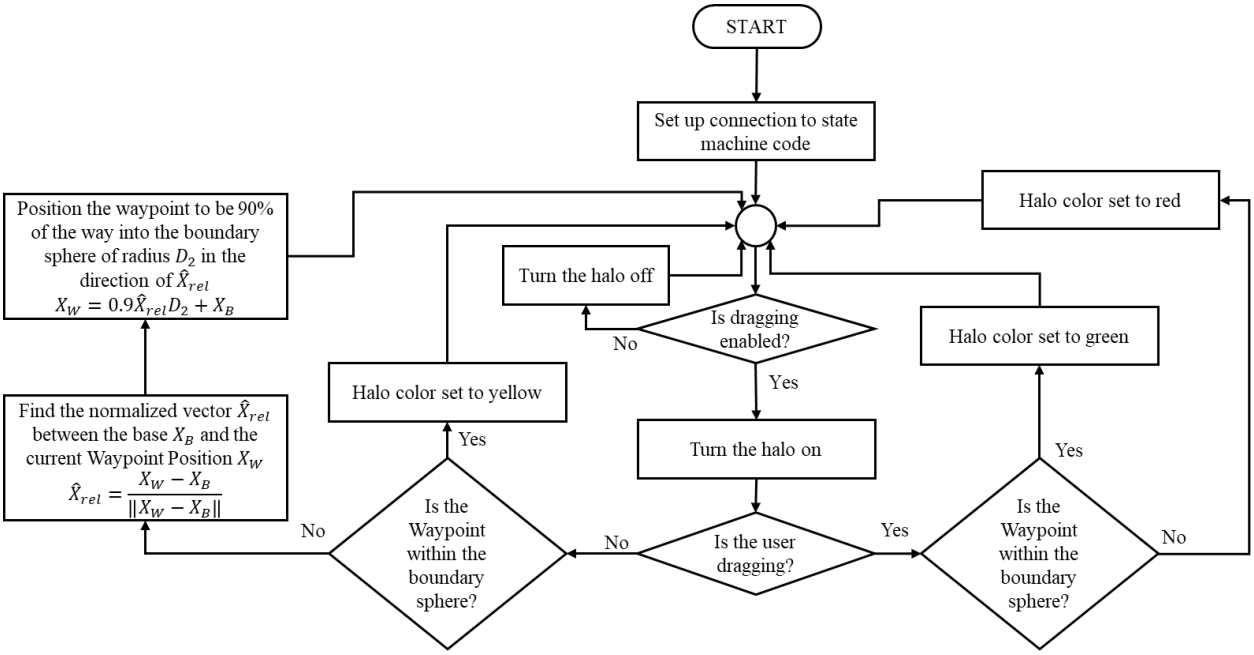


Figure 20 DraggingWaypointManager flowchart

2.3 Teach Pendant Modelling

The Teach Pendant is the handheld companion tablet that comes with a Universal Robots device. Through this, the user can control the robot's movements, as well as program it. Since this an essential part of the experience of using a collaborative robot, it was desirous to model the Teach Pendant for the design tool.

Figure 21 User jogging the UR robot tool flange through the Teach Pendant [16]

As the Pendant has the entire suite of functions that Universal Robots offers, it would be impractical to model every function exactly. Of those functions which are most important to a beginner user are motion instructions and the ability to teach the robot.

Before implementing a behavioral model of the Pendant in Unity, it was necessary to have a representative 3D model. Unfortunately, one could not be readily available online. By looking up images and estimating the size, a SOLIDWORKS model of the Pendant was made.

Figure 22 (Left) UR Teach Pendant inspiration, (Right) SOLIDWORKS model of the Pendant

This model was then imported into Unity, with the task at hand of creating a "screen" like the GUIs represented by the Teach Pendant. The GUI elements were made using a combination of Unity prefab items, as well as custom drawings of sprites in MS PowerPoint. Two tabs were made, "Motion" and "Teach" for fulfilling the two functions required to have a decent demonstration capability. Each tab deserves its own explanation as to its features and available functions.

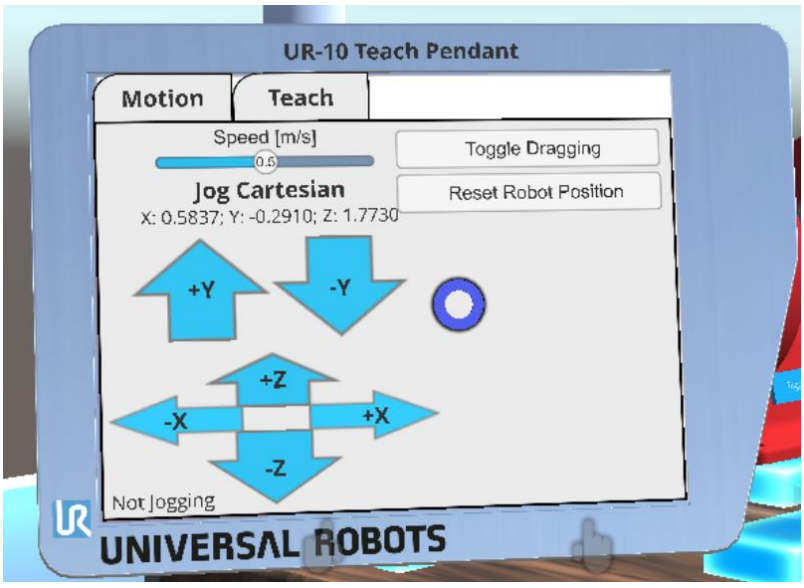


Figure 23 UR Teach Pendant Motion tab

The main purpose of having motion instruction options on the Pendant, as opposed to simply programming positions with hand movements, is to obtain precise motion. It made sense, then, to have the Jog instruction mapped to the simulated Pendant. “Jogging” in industrial automation refers to unconstrained movement in a given direction for an unspecified amount of time, but at a fixed speed. In the case of the UR-10, the jog commands work when the button is pressed. The user can press these buttons on the screen and have the tool flange move forward or backward in the X, Y, or Z coordinates of the world. For the user, these coordinates would correspond to forward/backward, up/down, and right/left, respectively from the point in which they instantiated the Hololens app.

To accomplish the Jog motion, the first step involves the GUI buttons detecting when the user was pressing down on the button. This was accomplished by creating a script called PointerListener, which responded to the OnHoldStarted, OnHoldCompleted, and OnHoldCancelled events. These events are called whenever a user is holding the button using the Tap and Hold Gesture. Each button is wired up to a different void method in the script RobotJog, corresponding to the direction requested. Figure 24 displays the flowchart for RobotJog.

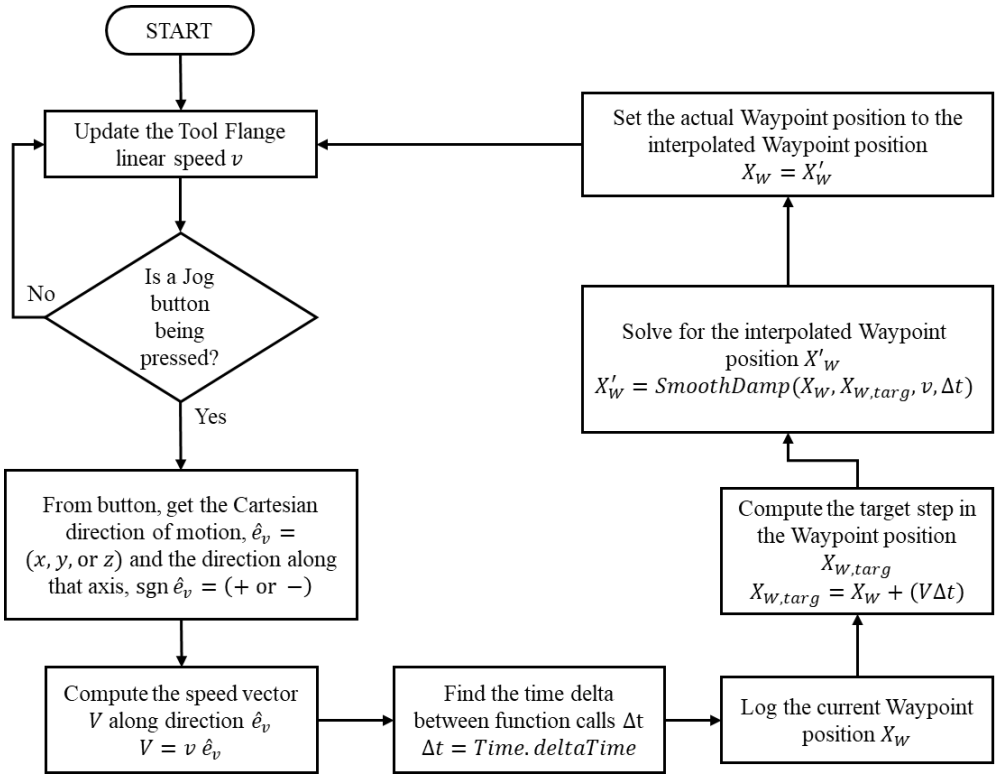


Figure 24 RobotJog flowchart

The process flow may appear unusually complex for such a simple function. For instance, one might assume that simply computing the interpolated position as $X'_W = X_W + (V\Delta t)$ would suffice. While it did move the Dragging Waypoint, and thereby the Tool Flange in the correct direction, the motion did not appear smooth. To fix this, the *SmoothDamp* Vector3 interpolator was used, as it was in the RobotMotionStateMachine_V3 script. Since the Waypoint object here

is the Dragging Waypoint, all the robust functions determining how and where the Waypoint has permission to be are functioning in parallel with RobotJog.

While jogging the Tool Flange in Cartesian space was modelled, jogging the Euler angle rotation of the Tool Flange was not implemented in time for the deadline of this thesis. It is highly suggested for future work to use RobotJog process logic in a similar way, but for rotation.

Other useful additions to the design tool are available as well through the Motion tab. The linear speed of the robot can be adjusted using the slider, which varies from 0 to 1 m/s. A printout of the exact XYZ coordinates of the robot is also available above the job buttons.

Two especially helpful features are the Toggle Dragging and Reset Position buttons. Toggle Dragging allows the user to switch between being able and unable to grab the tool flange. Normally grabbing is disabled, so the user would have to press this to enable it. The Reset Position button will relocate the Dragging Waypoint where the robot started in the scene, which is a nice spot near the center of the UR-10 workspace.

The second major function of the Teach Pendant is to teach the robot how to do things! This is essential for allowing the robot to perform tasks autonomously. Of course, there are many different functions a robot can learn, but again it would be time-prohibitive to program all of them. In Chapter 3, a single overall task is to be programmed for the robot. Therefore, the notion of a "routine" was formed, being a sequence of "actions" that one can "play" back, or "loop" repetitively. The most useful "actions" that could be modelled would be "moves" and "time delays".

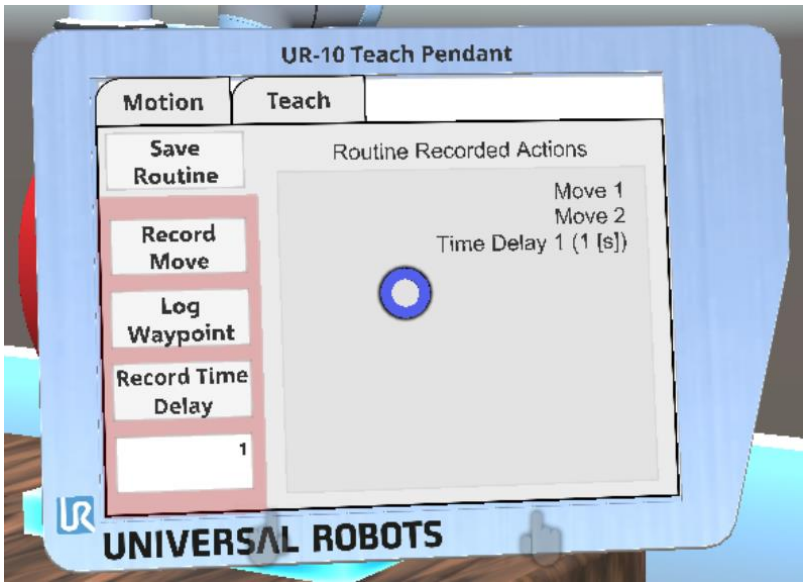


Figure 25 UR Teach Pendant Teach tab

A move is considered to be a sequence of logged waypoints, the same kind used in the robot's motion state machine. For example, a move in a square would consist of four waypoints logged at 90 degree angles from one another (five to move back to the starting position). The move is considered complete when the last waypoint logged in the sequence is reached by the Tool Flange.

The time delay is a counterbalance to the move, in that the robot simply waits in place for a specified amount of time. It's unlikely that in any application, a robot manipulator should be constantly bouncing around to different positions. In a CNC tending case, for example, the robot would have to wait until the part is done shaping before picking it up. While logging the waypoints is done through a button, the time delay is entered manually by the user in seconds through a separate virtual keyboard, containing a numeric keypad.

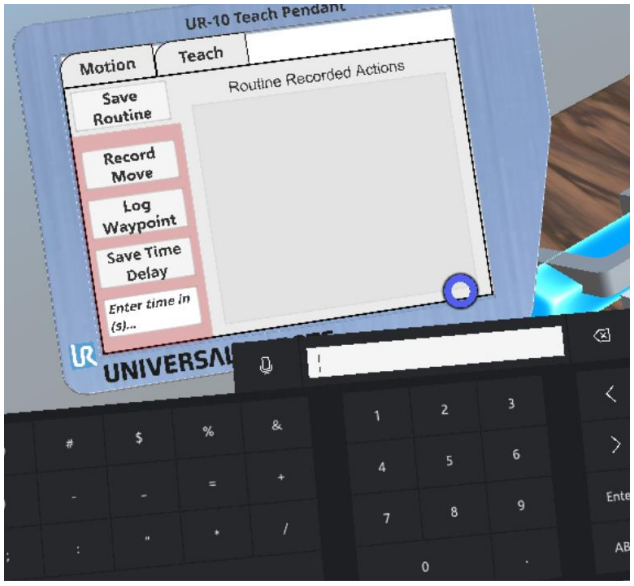


Figure 26 Typing in the delay time into the Teach Pendant

After saving the routine, the user may play it back a single time by clicking "Play Routine". This is useful to test if the teaching worked. However, turning on "Loop Routine" has a more practical application with repetitious automation. For the case below, Move 1 would run and the robot would wait 8 seconds before doing Move 1 again, and so on.

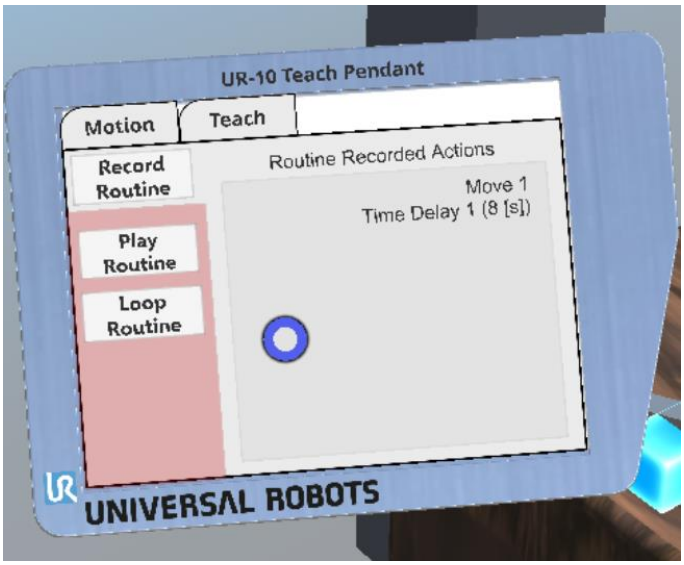


Figure 27 Teach tab after recording example routine

The logic of the Teach tab menu options, and the corresponding actions, are contained within the TeachTabGUIControls and RobotTeachV2 scripts. These two scripts are attached to the Teach Tab's Pane object for organization, although their functions are ambivalent to the GameObject they are attached to. While TeachTabGUIControls determines which buttons and windows should appear on the screen, RobotTeachV2 can read in these parameters and give commands to the Robot.

Before diving into the flowchart for the Teach mode, it is relevant to understand what data structures are used to save routine elements. Many of these structures, or "classes" in C#, contain properties which are Lists of various data types, T. Unlike an array, a List can change size dynamically, useful for adding any number of actions, moves, waypoints, time delays, and so on. Below is a table of useful C# List methods for the reader's reference.

Table 4 Important C# List methods

High-level function	C# List method
Instantiate a variable var as a List of data types T	var = new List<T>;
Append a new element, foo, to the end of var	var.Add(foo);
Determine the number of elments in var, bar	bar = var.Count;
Obtain the n^{th} element of var, thing	thing = var[n-1];

The order and type of actions of a routine is specified in a TeachRoutine class. This class has two properties, ActionTypeList and LookupKeyList, which are both C# Lists of type int (integer). The ActionTypeList holds the sequence of action types that were logged in the Routine. Each action type is given a number: 0 for "moves", and 1 for "time delays", with the flexibility to add more types of actions in future work. The LookupKeyList holds the index for each action's list. Say the current action is the fifth action in the routine sequence, ActionTypeList[4], and it is a "time delay". If the corresponding time delay is Time Delay 3, then the LookupKeyList[4] = 2.

As alluded to above, the recorded moves and time delays are recorded in separate Lists, named moveList and timeDelayList, respectively. To start with the easiest one, timeDelayList is a List of type float, with each element representing the collection of time delays logged by the user in seconds. To connect with the example above, if the third time delay logged was 3.5 seconds, then timeDelayList[2] = 3.5. The moveList is more nuanced due to the complexity of defining the data hierarchy of a "move". The underlying data type of moveList is a TeachMove, which is a class defined itself by and equally-long positionList and rotationList. The two lists update concurrently to record the Transform data of the logged Waypoint sequence. Thus, the positionList is a List of type Vector3 and the rotationList is a List of type Quaternion.

With this background information, two flowcharts are presented. The first is Figure 28, which is the "recording" function, starting with "Record Routine" and ending with "Save Routine". The second is Figure 29, which is the "playback" function, starting with "Play Routine". It is expected that the user does recording first, and playback second.

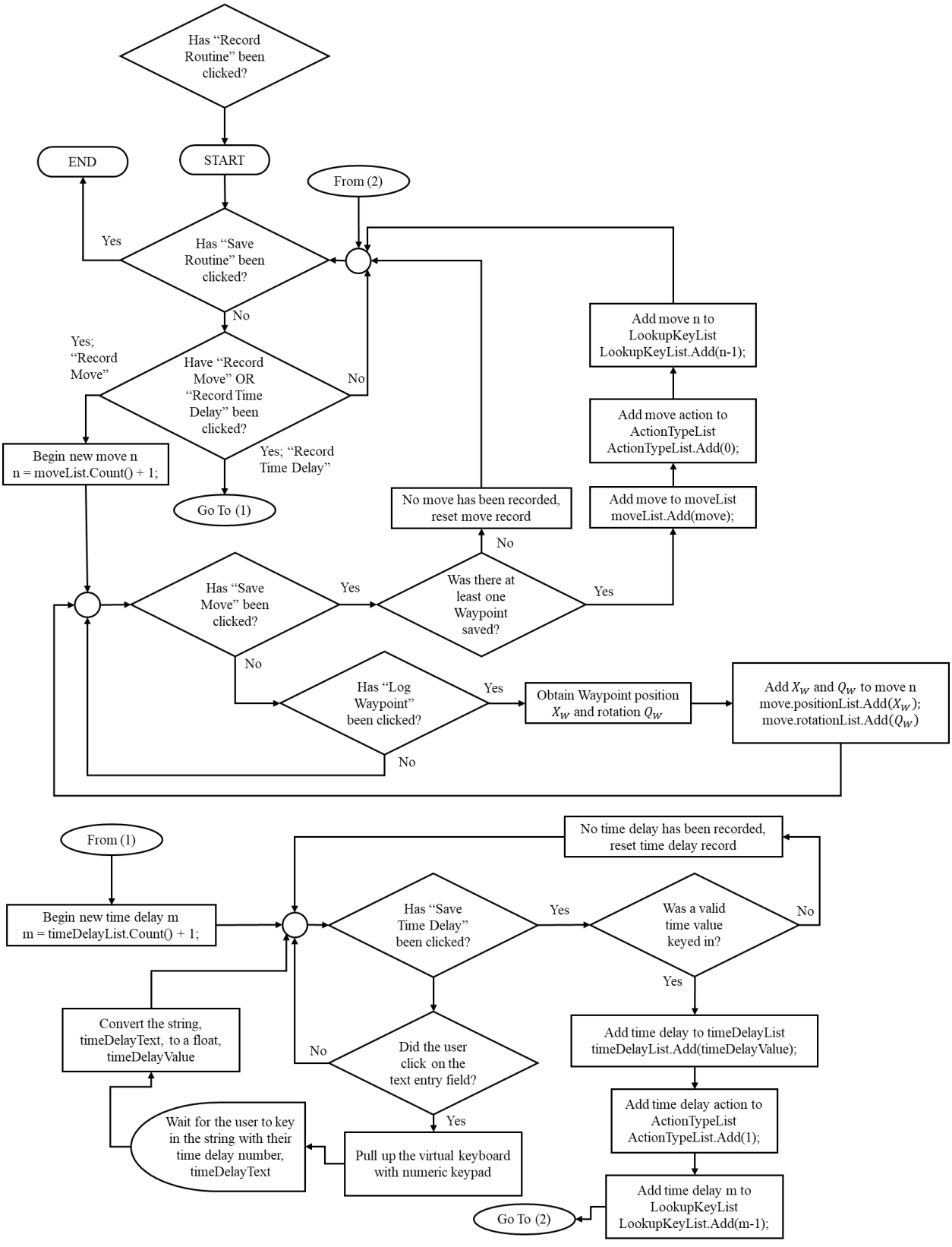


Figure 28 Teach mode, routine recording flowchart

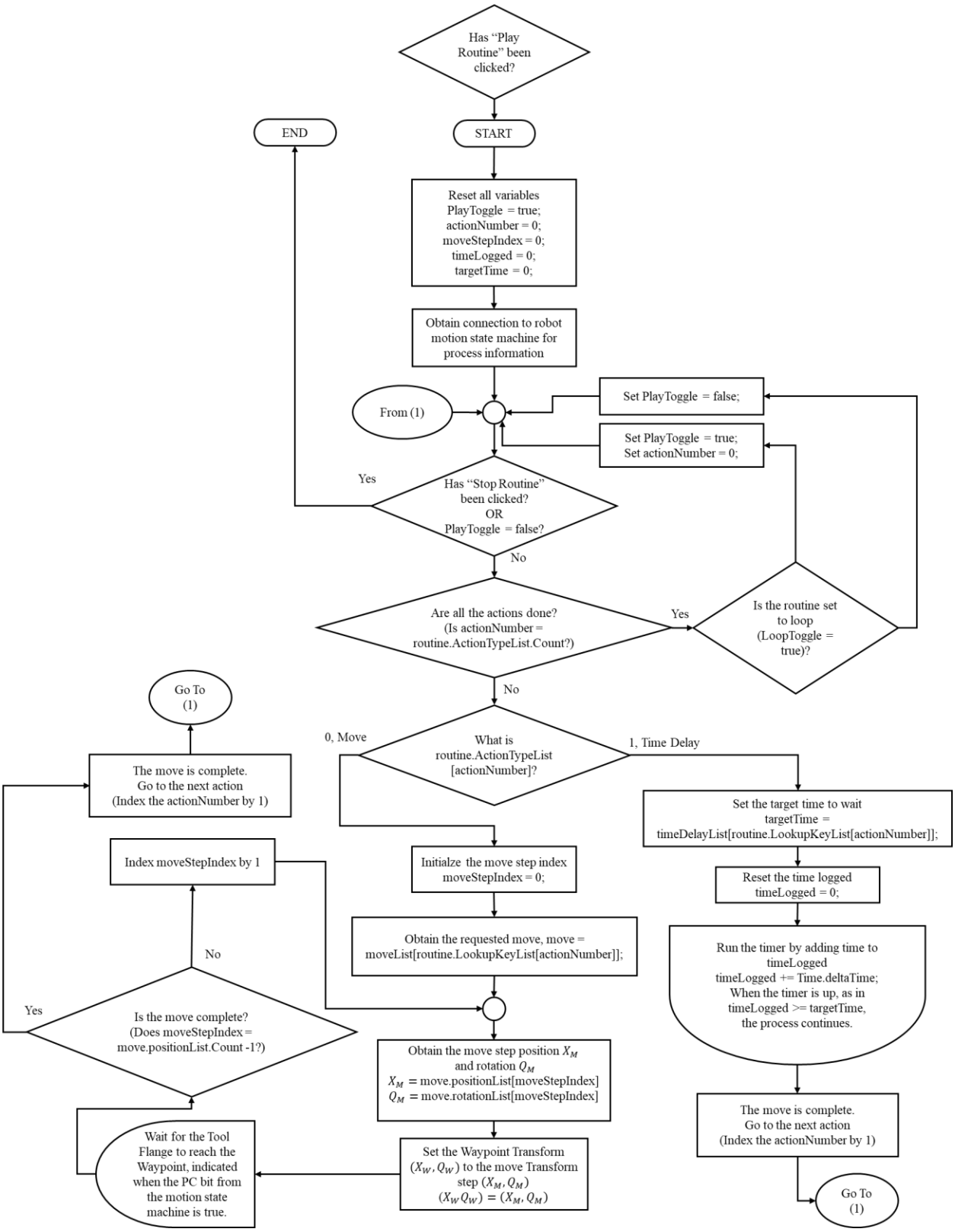


Figure 29 Teach mode, routine playback flowchart

One non-technical issue with the Teach Pendant initially was that it was necessary to have near the user, but difficult to keep a hold of. Having the Pendant close to the face enough to see the screen details, but far enough away to have the Gaze work, was a fine balance to strike. Furthermore, it would be cumbersome to have to carry around the Pendant throughout different parts of the scene.

To resolve this, a script was developed to move the Pendant around with the user and have it stare at the user. For this, the Teach Pendant GameObject was made a child of an invisible cylinder GameObject, which we will call the "Menu Carousel". The Pendant was put on the circumference of the Menu Carousel, so that when the Carousel rotates, the Pendant moves in a circle.

The Menu Carousel has a script called CarouselMenuController, which performs the function described above. The LookRotation and ProjectOntoPlane functions are native Quaternion and Vector3 methods, respectively. The calculations in Table XX occur during the Update loop of CarouselMenuController.

Table 5 CarouselMenuController Update loop operations

Step	Equation
1. Set the menu carousel position X_C to the base of the robot X_B (usually only useful if the robot moves)	$X_C = X_B$
2. Find the vector X_{PU} in the direction pointing from the pendant's position X_P to the camera/user position X_U	$X_{PU} = X_P - X_U$
3. Find the quaternion Q_{PU} that describes rotating the pendant towards the user	$Q_{PU} = LookRotation(X_{PU})$
4. Find the vector X_{CU} that describes pointing from the menu carousel's position X_C to the user position X_U	$X_{CU} = X_C - X_U$
5. Flatten the vector so it only has coordinates in x and z by projecting it onto the y-normal plane	$X'_{CU} = ProjectOntoPlane(X_{CU}, Vector3.up)$
6. Find the quaternion Q_{CU} that describes rotating the menu carousel, along the y-axis only, towards the user	$Q_{CU} = LookRotation(X'_{CU})$
7. Set the rotation of the carousel Q_C, which places the Pendant close to the user	$Q_C = Q_{CU}$
8. Set the rotation of the Pendant Q_P, so that it tilts to face the user	$Q_P = Q_{PU}$

Additionally, the Pendant's metal surfaces were given a Collider, and the HandDraggable script was attached to it. This allows a user to pull or push the Pendant to where it is convenient for them at the time. Assuming the user was looking in the direction of the robot's base, the Menu Carousel will maintain the updated relative position of the Pendant as the user walks around the robot. Figure 30 demonstrates these capabilities towards making the Pendant more accessible to users.

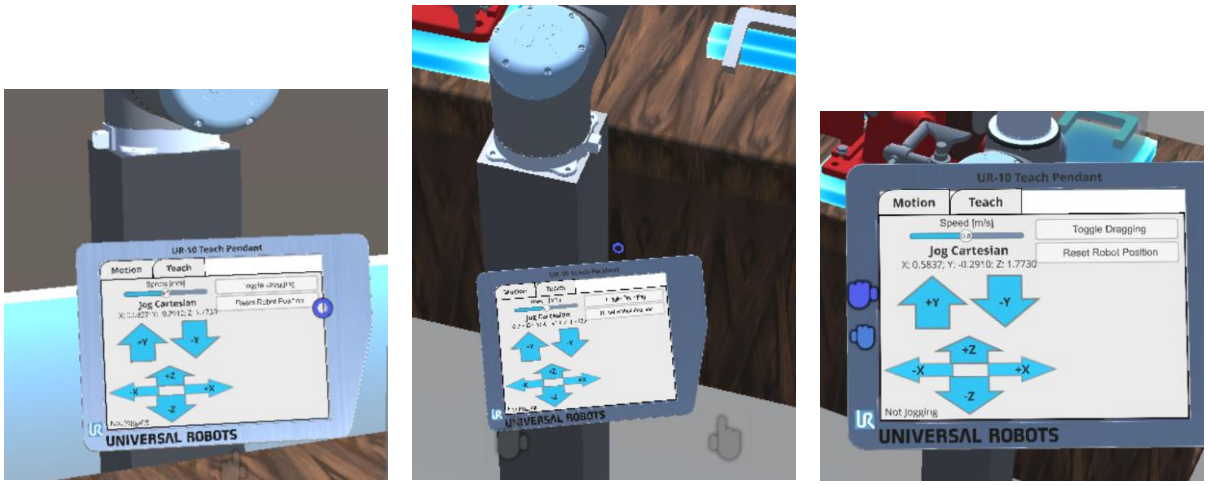


Figure 30 (Left) Teach Pendant where it starts in the scene, (Middle) Pendant circles around and tilts up to face the camera/user, (Right) user drags Pendant closer to see screen GUI elements

In summary, many features and improvements were made in modelling the cobot, and the Teach Pendant was a new addition. With these core elements at a satisfactory level of completeness, it follows that a manufacturing scenario be developed to use the design tool in a practical setting.

Chapter 3: Development of the Manufacturing Scenario

To demonstrate the ability of the cobot model to be used in a collaborative way, a manufacturing scenario needed to be selected. Once one was selected, it needed to be modelled in such a way as to incorporate a sense of realism and interactivity. The following sections detail what was chosen, and how it was implemented.

3.1 Selected Manufacturing Scenario

The manufacturing scenario that was used as an example is a grinding and polishing operation. Imagine a small firm that machines custom metal objects, say door handles. After some CNC or milling work, the work piece may have remaining shaping and finishing steps before it is ready to be packaged or sold. A door handle, in example, may have the generally correct shape but needs to trim down the hand-grasping part. While CNC may be able to get the geometry right, grinding is a far simpler and faster shaping process. Door handles may have decorative value as well, so a polishing operation would increase the quality.

Grinding inherently poses health and safety hazards. It nearly always creates sparks, noise, and vibrations, all of which can be absorbed by the manufacturing associate. Furthermore, grinding wheels pose a hazard to the hand, due to the abrasive and fast-spinning wheel. Although appropriate PPE can protect against these hazards, it is a dull and dangerous job. Polishing, on the other hand, is safer and requires more skill and attention to detail than grinding.

This manufacturing scenario has the right elements that make it good for human-robot collaboration. First, the robot and the human would be better suited to grinding and polishing, respectively, for the reasons listed above. Second, the cycle time for both operations is comparable. Third, the two operations follow each other sequentially in the manufacturing process. Finally, the handling of parts between processes occurs in the same space and uses the same machinery. While these are not all necessary conditions for supporting human-robot collaborations, together they are certainly sufficient conditions to consider it a good idea.

To build out this scenario, a bench grinder and a handle had to be modelled for the purposes of the manufacturing process. The bench grinder looks and operate as it does in real life, with a grinding wheel on one side and a polishing wheel on the other. The handle was modelled to simulate deformation during grinding, and changes in appearance during polishing. Careful placement of the cobot and manufacturing associate in the work cell ensures that the two can work within the same space around the work bench, but that the cobot is able to move freely without being in the way of the associate. The work cell layout is presented in Figure 31 below.

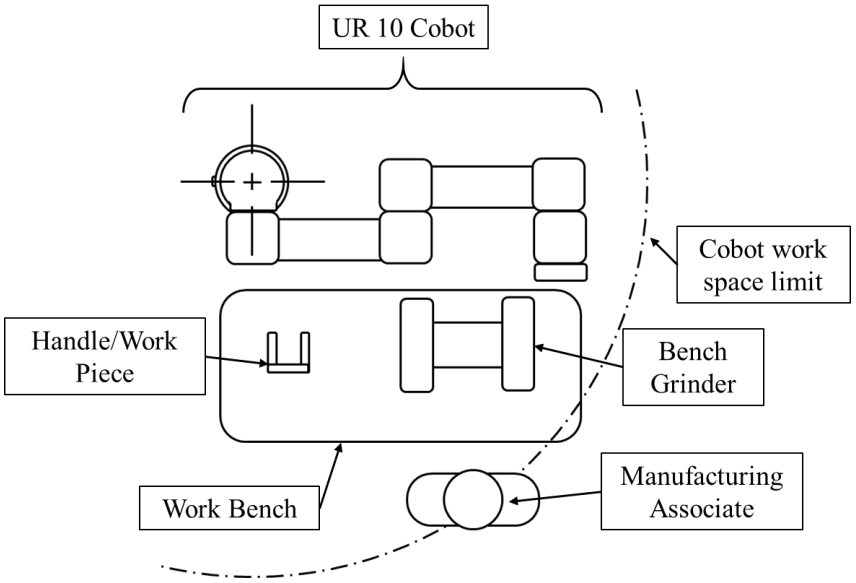


Figure 31 Manufacturing scenario work cell layout

1.2 Bench Grinder Modelling

To begin, a model of a bench grinder was found on GrabCad as a SOLIDWORKS assembly file. While the picture rendering looked very nice, the SOLIDWORKS model lacked any color, so the model was colored scarlet. To keep sparks away from the worker, the cover for the grinding wheel was flipped to the opposite side of the bench grinder. The grinding wheel is colored brown and the polishing wheel is colored gray.

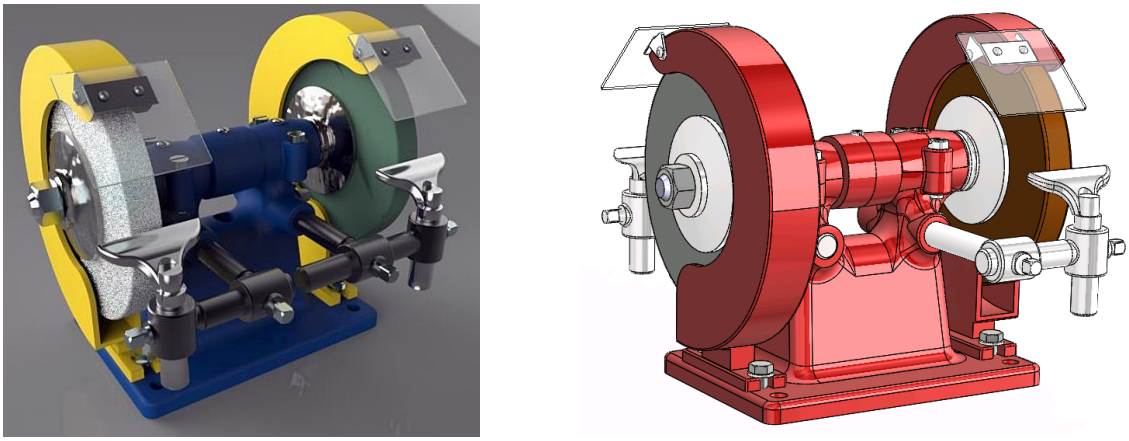


Figure 32 (Left) Bench grinder rendering from GrabCad [17], (Right) Modified Bench Grinder

The SOLIDWORKS model contains many, many parts which can be summarized functionally as either being rotating or static. Therefore, two different SOLIDWORKS part files were constructed to represent these two conditions.

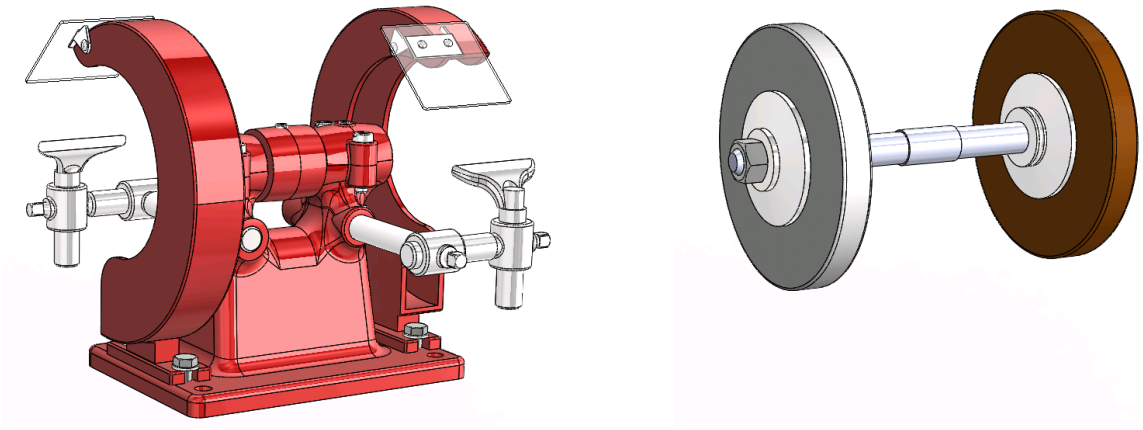


Figure 33 Bench grinder static (left) and rotating (right) separated solid bodies

The solid objects were modified slightly in 3ds Max after learning that the materials could be modified there, for example to create transparency effects. In Unity, the materials of the grinding and polishing wheels were further modified by overlaying an albedo texture map of gravel and carpet, respectively, to give off the appearance of sand grit and fibers. The rotating and static objects of the bench grinder were connected through a Hinge Joint, so that the rotating part could spin within the static part. The Hinge Joint happens to also have a "motor", which can rotate the connected mass at a target velocity with a certain available torque or what it calls "force" (no units given). The motor was given a target velocity of 500 deg/s, which is 3000 RPM, and a "Force" of 90. To give the rotating body a sense of inertia and damping, its Rigidbody component parameters for mass, drag and angular drag were set to 20, 0, and 0.5. The motor also is given a rev-up time of 3 seconds to ramp up the force from 0. A toggle button was added to turn on and off the motor.

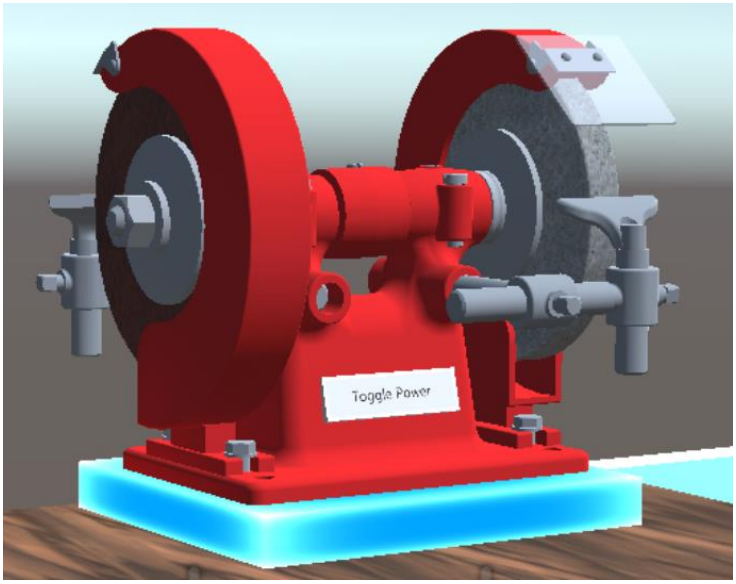


Figure 34 Bench grinder in Unity, with the polishing wheel-side facing the user

Besides modelling the solid and motion behavior of the bench grinder, some creative liberty was taken in generating realistic sound and visual effects. Generic motor sounds were found online and

imported into Unity. The pitch and volume of the motor was set to be proportional to the angular speed of the Rigidbody component.

Grinding metal also carries an expectation of noise and sparks. Another sound was added from generic sounds of grinding against metal. The sparks were generated through the Particle System component, one for directional sparks down and another for random sparks. When a handle is brought to the grinding wheel, the sound and sparks appear if the grinding wheel is rotating near its operating range. All of this is contained within the BenchGrinderEffectsManager script.

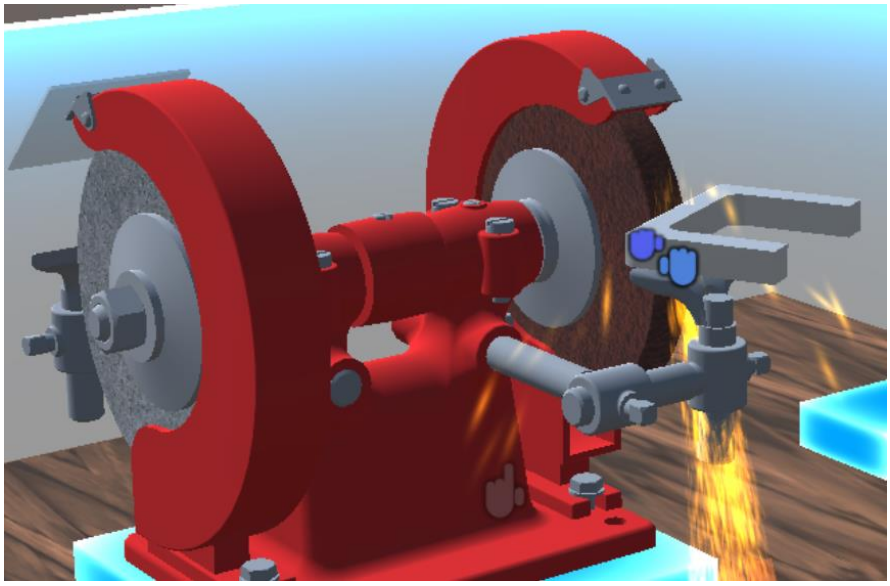


Figure 35 Handle generating sparks as it is in contact with the grinding wheel

3.3 Handle Modelling

A handle was a good sample metal work piece for this application, since it is geometrically simple yet amicable to the proposed grinding operation. All one would have to do to demonstrate the deformation of the handle would be to fillet the part where grasping occurs. A handle was made in SOLIDWORKS that was completely unshaped, and then four parts were made identical in every way except for the fillet size. The fillet sizes ranged as well in increments of 25%, 50%, 75%, and 100% of the final size value. The different models are presented in Figure 36 below.

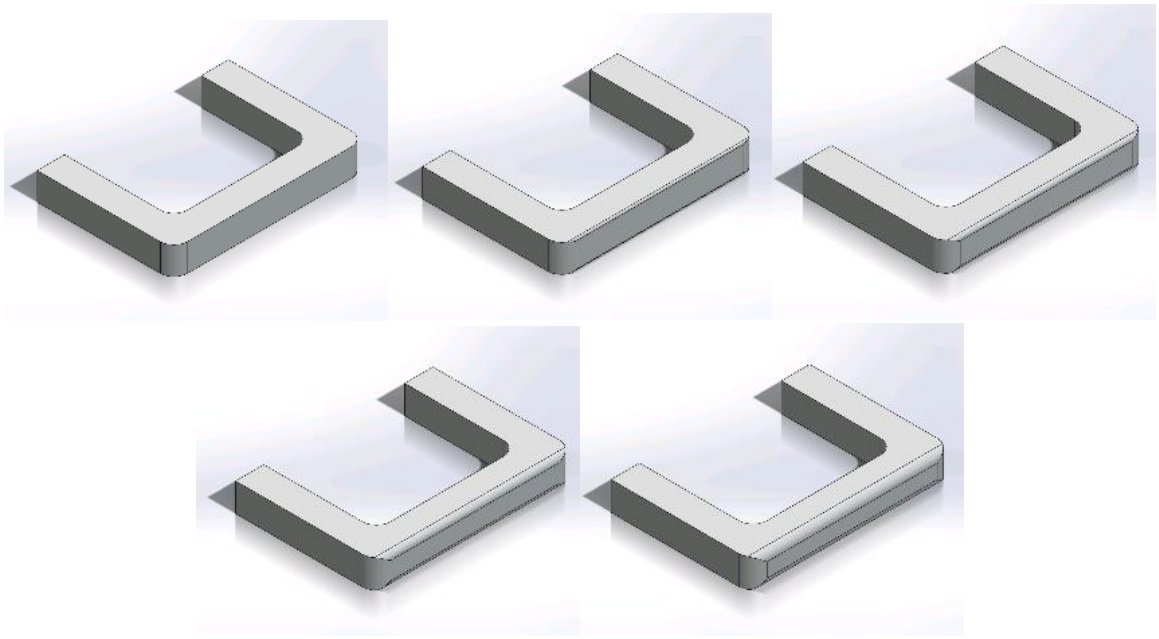


Figure 36 Progressive shaping of handle fillet at 0%, 25%, 50%, 75%, and 100% shaped

Each of these objects was independently imported into Unity. For purposes of coding, it was desirable to keep only one GameObject but have multiple different appearances for it as it went through the grinding process. The HandleMeshSwapper script attached to the Handle GameObject tracks for how long the handle has been "grinding", or how long it's been colliding with the grinding wheel with the speed high enough. For every quarter of the time it takes to finish grinding, the script substitutes the old Mesh in the MeshFilter component of the Handle for a Mesh that came from a more shaped handle. In example, if it takes 10 seconds to grind the handle, then every 2.5 seconds the MeshFilter will receive a new Mesh. The meshes are named "Unshaped", "Pc25", "Pc50", "Pc75", and "Pc100" following the order of the solid models presented in Figure 36.

It was convenient to re-use the HandleMeshSwapper script to also update the graphical properties of the handle after polishing. We assume that the GameObject's Material needs to change in order to have it appear brighter than it was before. As with the grinding timing, polishing with the polishing wheel is also timed for so many seconds before it has been "polished". At this time, the MeshRenderer component of the Handle GameObject had its material changed to something lighter. The HandleMeshSwapper process flow diagram is displayed in Figure 37 below. It should be mentioned that the grinding and polishing state recognition comes from the BenchGrinderEffectsManager script.

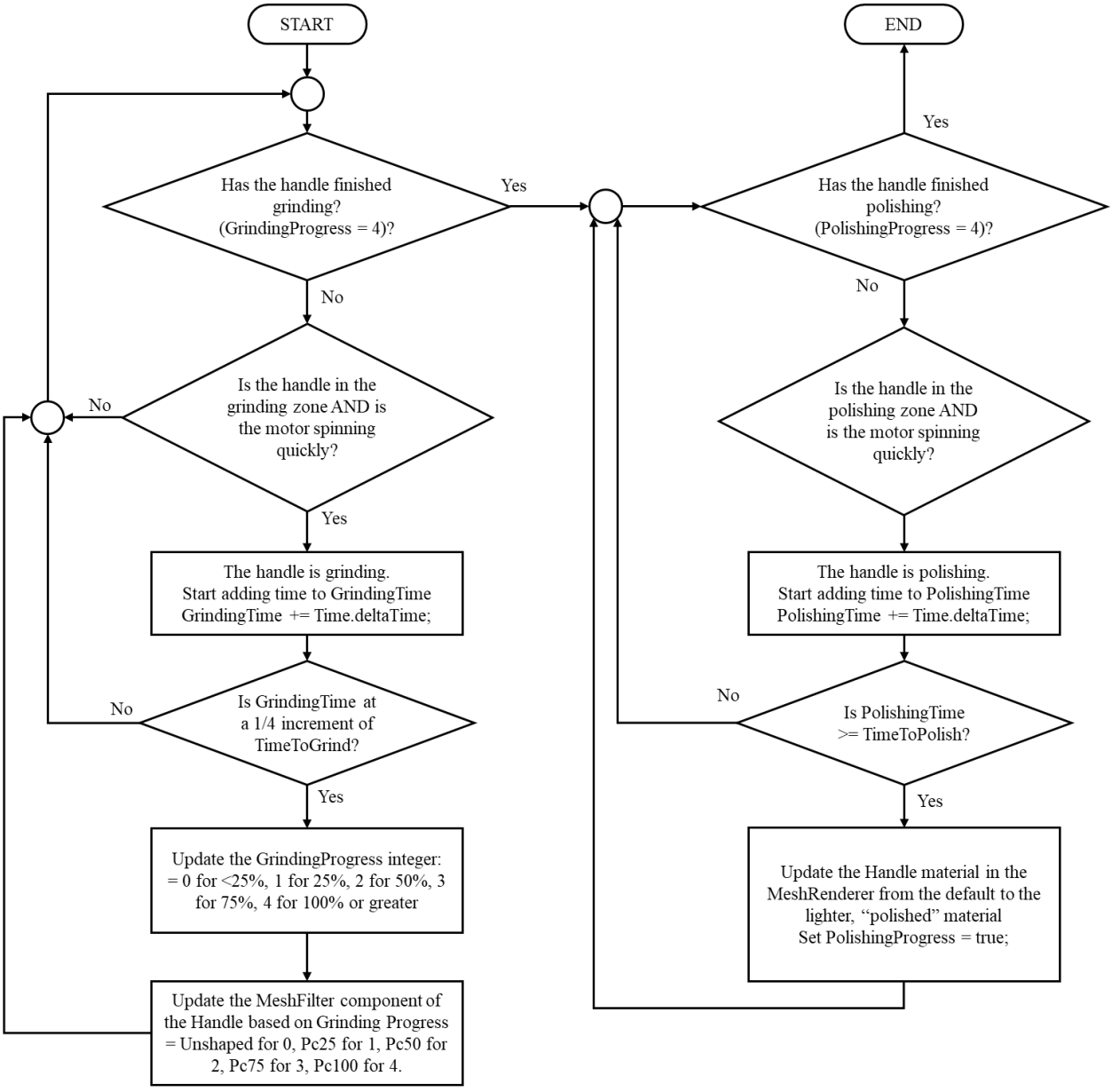


Figure 37 HandleMeshSwapper flowchart

3.4 Task Sharing and Interactivity Improvements

It is expected that the cobot and human manufacturing associate share in the overall task of finishing the handle, with the cobot and associate performing the grinding and operations, respectively and in parallel with one another. For them both to succeed in this cooperation, additional facilitation between the two should be supported. Some logical questions may arise, like: How will the cobot when to pick up to grind the handle, and drop it off after grinding it? Where will, or should, the drop-off location be? Likewise, where should the associate pick up the handle after it grinding, and where should they place it after it is finished? Furthermore, how can this process be repeatable to truly simulate the efficiency of human-robot collaboration?

Before diving into the implementation, a better understanding of the finishing task from start-to-finish was required. We assume that the robot has already been programmed, and that there is an infinite supply of handles waiting to be finished.

1. The cobot picks up an unfinished handle at a location on the work bench.
2. The cobot moves the handle over to the grinding wheel
3. The cobot takes some time to grind the part, say 15 seconds.
4. The cobot moves the handle to a designated "ground" drop-off place on the work bench
5. The manufacturing associate takes the ground, but unpolished part, and moves it to the polishing wheel. Meanwhile, the cobot has already begun to execute steps 1 through 4.
6. The associate takes some time to polish the part, say 10 seconds.
7. The associate moves the polished part to a designated "polished" drop-off place on the work bench.
8. By this time, the cobot should have completed steps 1 through 4, and so the associate may repeat steps 5 through 7. The process continues.

The common denominator between all these steps is the handle, as it is the focus of the finish operation. Additional scripting was required to support the placement of the handle. Also, the ability to spawn handles for the cobot and to delete handles when the associate was complete was desired. Finally, the drop-off locations needed to be created and give a visual cue to the user.

First, the handle placement for the robot and the grinding and polished drop-off locations were modelled. These can be seen in Figure 38 below, from left to right. The drop-off locations have a transparent box with a hologram model of a handle through them, and descriptive text. Notice how the ground handle drop-off is nearest the grinding wheel, whereas the polished handle drop-off is nearest the polishing wheel. This gives the cobot and associate their own work spaces to perform their duties. While a handle is placed at the start to the far left, a button with "Spawn Handle" will allow users to spawn another one if they drop or lose a handle.

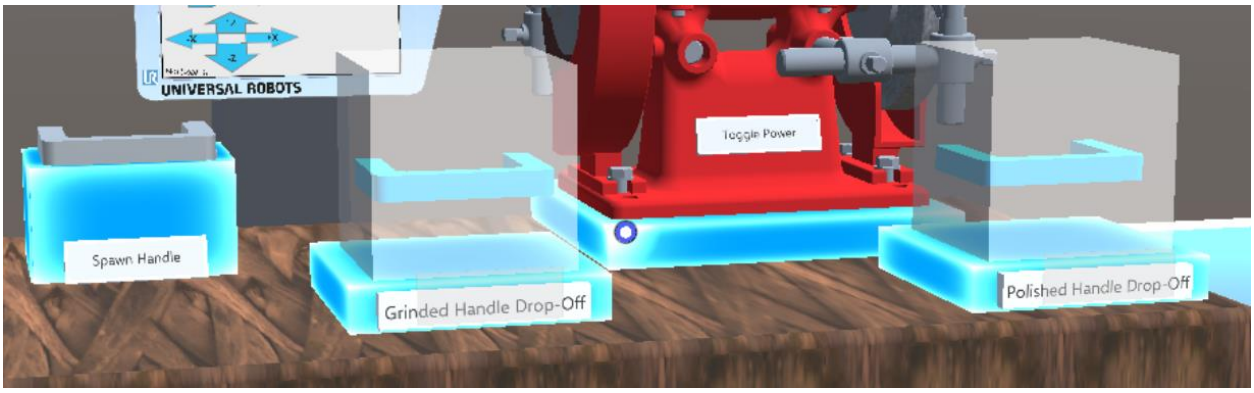


Figure 38 Handle spawn location, ground and polished handle drop-off locations

Most of the functions required to spawn and destroy the Handle object, have the cobot Tool Flange be able to pick it up, and place it in an appropriate drop-off location, are all managed by the HandlePlacementController script. The script borrows the grinding and polishing state information from the HandleMeshSwapper script to manage the placement in appropriate steps of the task.

Figure 39 has the flowchart of what the script does. It should be noted that when a new instance of the Handle is spawned, the script will begin at the "START" point. Unlike previous versions of the flowcharts, which were heavy on code and transformations, this flowchart is more about the task flow of the handle as it goes through the finishing process.

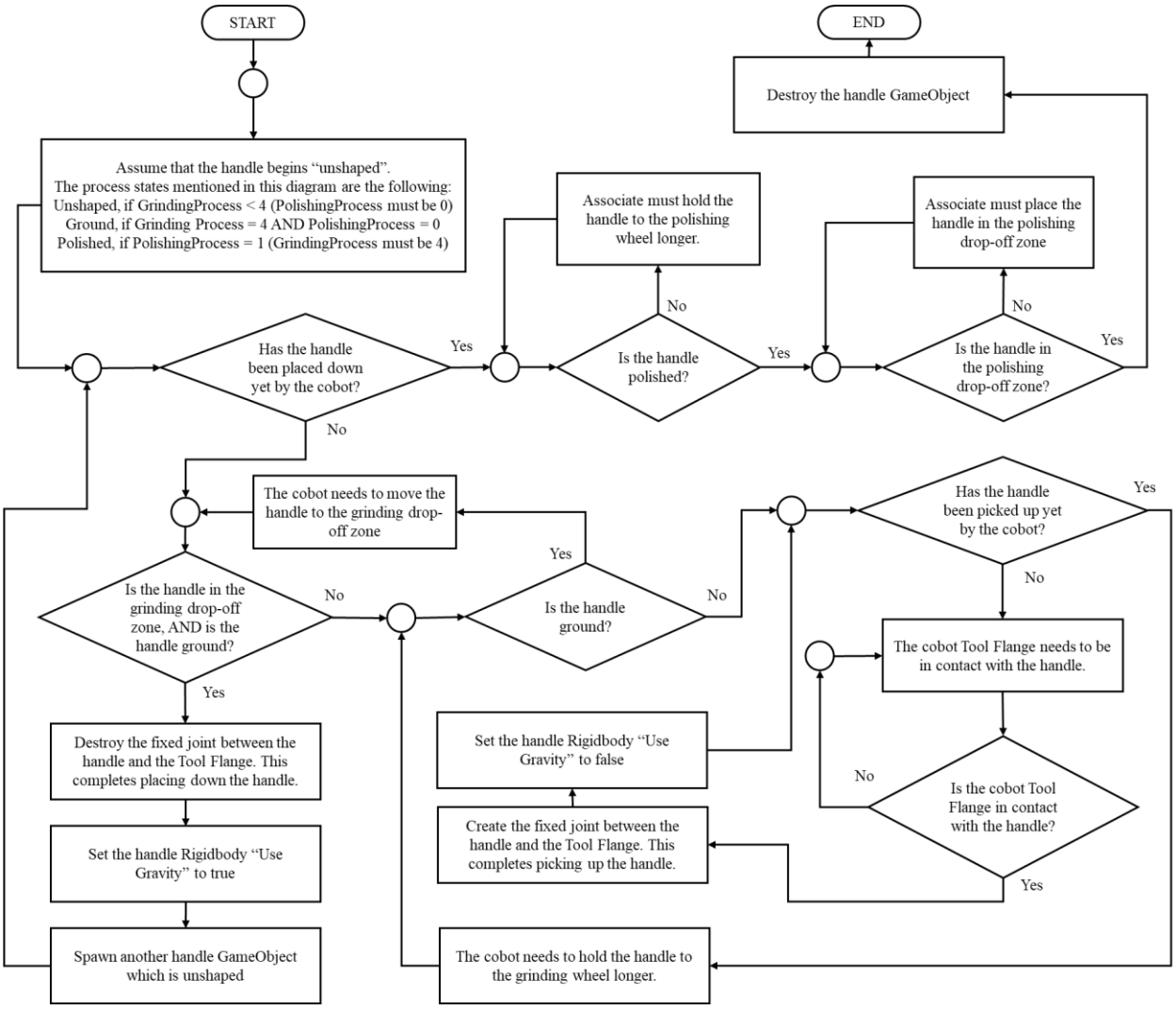


Figure 39 HandlePlacementController flowchart

Besides the placement of the handle, the placement of the rest of the elements in the manufacturing scenario was desired. The blue base on GameObjects is a cue that they can be repositioned in the environment, with the TapToPlace script from the Mixed Reality Toolkit. When they tap the blue base, the base along with the object it is attached move with the user's gaze. While the position of the scene elements should be sufficient anyways, it offers the user flexibility to reposition items around the environment.

Chapter 4: Testing the Tool

Throughout the process of building the design tool, it was periodically uploaded as an app to the Hololens to test it. For example, dragging the Tool Flange was much more intuitive using the Hololens than the emulator in the Unity editor. However, with the tool completed to a point where the manufacturing scenario can be fully represented, it was time to test it out and see how well it worked.

As with everything on the Hololens, the app must be launched from the Start UI. If it is not directly pinned here, then it can be found on All Apps through the "+" button. The app title is the title of the Unity project – in this case, it was MotionPlanning. Clicking on the App allows one to place the UI pane anywhere in the environment. After it has placed, the app will start. The splash screen says "Made with Unity" – this is a cue that the app is loading. Since the environment and scripts are rather complex, the Hololens takes a couple of minutes to load the scene.

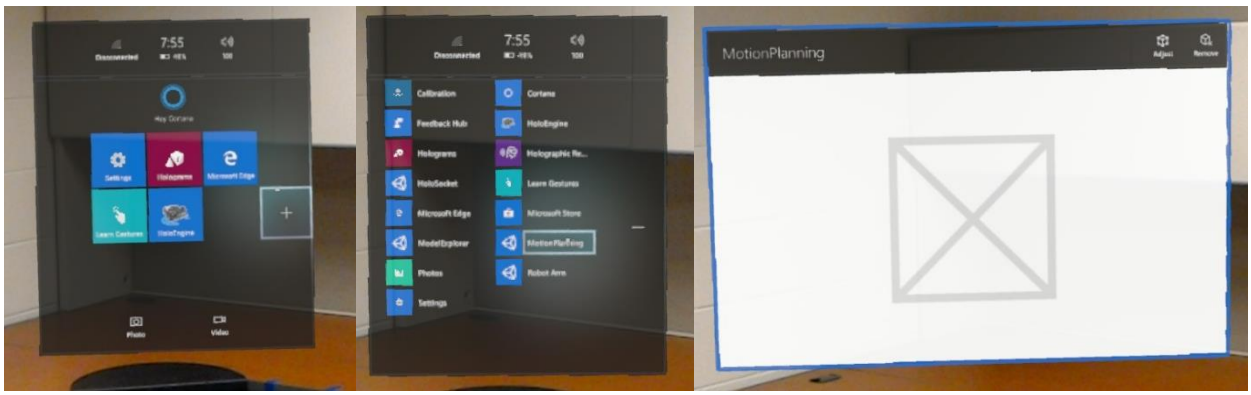


Figure 40 Navigating to the design tool Hololens app (note – zoomed in from full person FOV)

When the scene is loaded, the user can look down to see the environment. This initial position for all holograms is set in Unity, relative to the world origin. For the holograms to be placed below and in front of the user, they are placed lower in Y and higher in Z, respectively. This version of the app happened to have the holograms were placed a little too close to the user. Despite this, most of the scene is visible right away as in Figure 41.

Also of note, the spider web-like mesh around is the Spatial Mapping tool trying to figure out the geometry of the room. Since our holograms were placed in a mostly empty room, it doesn't bear much impact on the function of the tool.

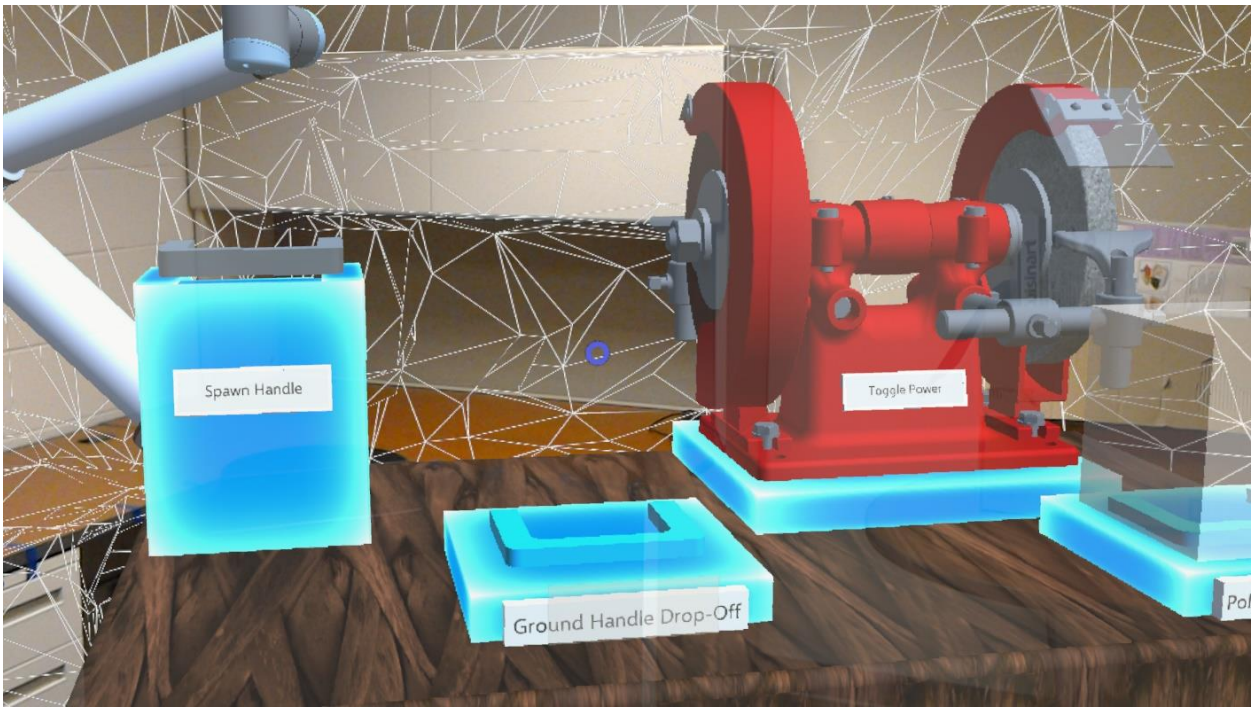


Figure 41 Scene holograms appear in front of the user.

To begin collaborating with the robot, the user is bound first to teach it the tasks it needs. Using the Teach Pendant, the user can select “Toggle Dragging” to better see the Dragging Waypoint and to move the cobot around. They should then press “Record Routine” to begin teaching the cobot.

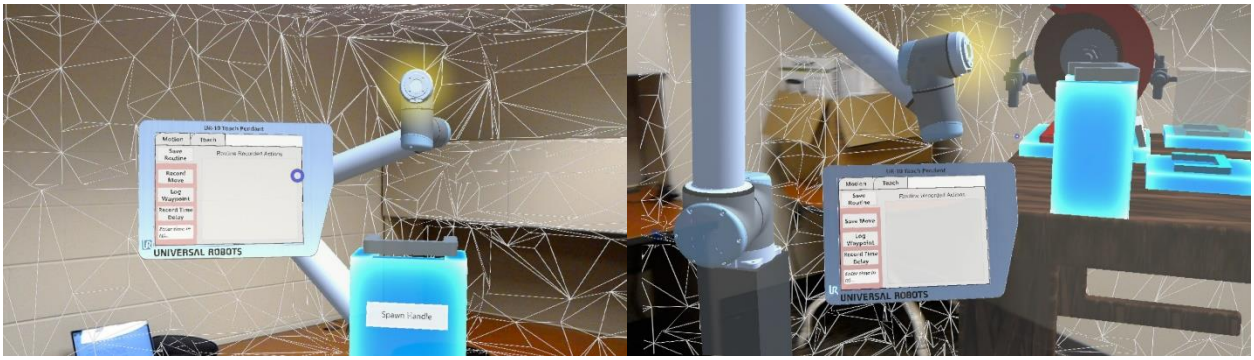


Figure 42 (Left) Starting a new move, (Right) coarse positioning near the handle spawn

They should log a Move such that the Tool Flange touches the handle, picking it up. To do this, they can combine the coarse movement of dragging the cobot close with the fine movement of getting the robot in position with the Jog feature. Each time, they should log a Waypoint. The final waypoint they log can be anywhere in a neutral position, to leave the cobot stable for the next step.

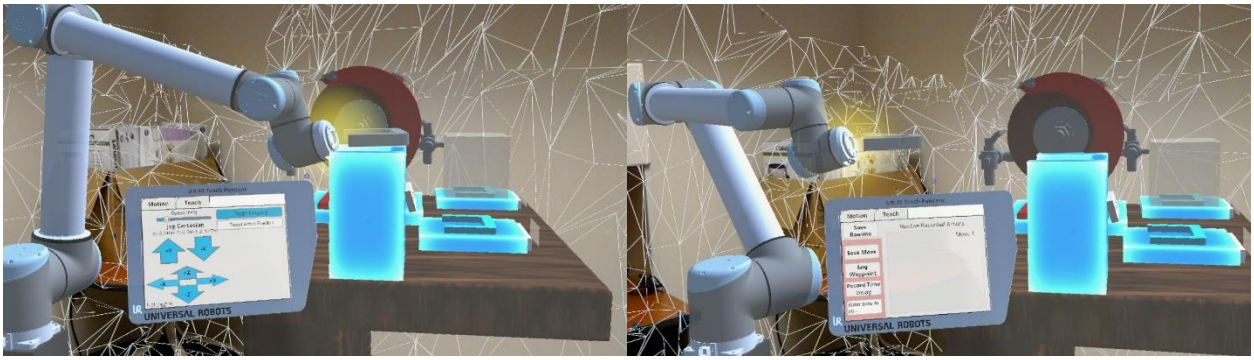


Figure 43 (Left) Fine positioning of tool flange to the handle, (Right) moving the handle to a neutral position before logging the next move

Next, the user should push the Toggle Power button on the bench grinder. This will spin up the motor, allowing the user to grind and polish. They should leave this on until they are finished prototyping the operation.

With the bench grinder on, the user should log another Move such that the handle is against the grinding wheel, and they can visually see that the handle is being shaped. As to not have the cobot clip through the bench grinder sideways, it is recommended to log an additional waypoint before grinding the handle that is further back. This allows the handle to enter the grinding wheel straight forward.

Figure 44 (Left) Logging a waypoint behind the grinding wheel, (Right) Handle making sparks on the grinding wheel (spacing given to avoid Rigidbody collision)

From here, the user can leave the handle grinding while they program a Time Delay. The time it takes to grind is 15 seconds, so the user should enter that into the Teach Pendant. To remind the reader, the user would select “Record Time Delay”, click the text field, enter “15” then “Enter” on the virtual keyboard, and then select “Save Time Delay”.

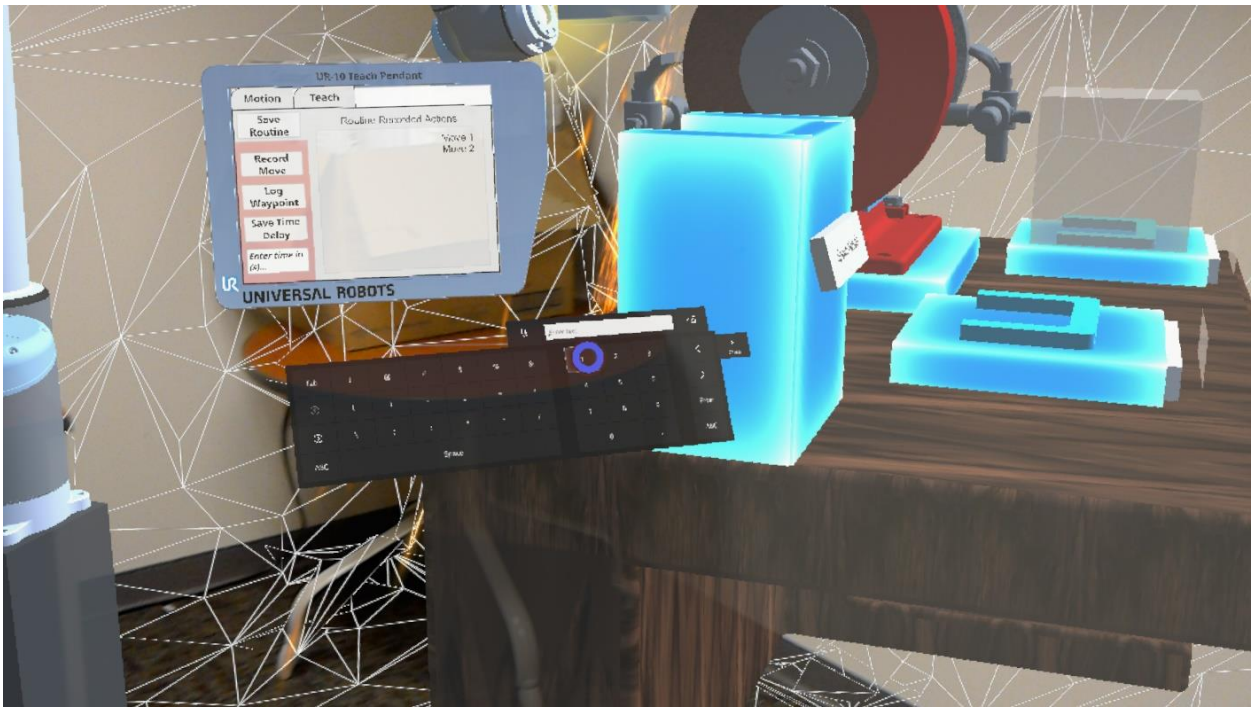


Figure 45 Typing in the Time Delay value on the virtual keyboard numeric keypad

Finally, the user would need to program one final Move to have the robot set down the handle. As before, this should consist of two moves, one to clear the bench grinder, and one to position the handle at the "Ground Handle Drop-Off". If successful, the handle will release from the cobot Tool Flange and will be placed onto the drop-off spot. Additionally, a new handle will have spawned to replace the old one for the robot. The user can then finish by pressing "Save Routine".

Figure 46 (Left) Logging a waypoint for the last move, before the drop-off zone, (Right) Dropping off the handle at the zone, a waypoint must be logged here

Now that the Routine is saved, the user can practice polishing the handle. The user should drag the handle over to the polishing wheel and hold it there for 10 seconds. When the handle is done, they will see the handle become a brighter gray. After they are done polishing, the user simply drags the handle to the drop-off zone next to the polish wheel, and the handle object is destroyed.

Figure 47 (Left) User holding ground handle to the polishing wheel, (Right) The color lightens after polishing – hard to see here, but easier in Hololens

With the process down the user can now practice the manufacturing scenario with the cobot. To try it out one time, the user can press “Play Routine” on the Teach Pendant and watch the cobot perform the grinding operation. If the user wishes to test out their speed, then they can select “Loop Routine” and “Play Routine” so that the robot will perform the grinding operation as quickly as it can. Likewise, it is expected that the user take the ground handle and perform the polishing operation. Based upon the bottleneck of grinding, no more than four handles per minute can be finished. However, the user could take extra time in moving or positioning the handle, so it is possible that the productivity is limited by the operator. If they ever feel behind the cobot while it is looping, they can press “Stop Routine” to finish the work they have piled up.

Chapter 5: Conclusions

It is appropriate that, with the development of the design tool at a point of sufficient completion, that the contributions made thus far be considered. The contributions given to the tool will be put up to the expectations of the design tool, per the objectives. Therefore, some claims can be made for describing the effectiveness of the tool. Finally, areas of future work will be discussed in how to improve simulating human-robot collaboration with VR/AR technology.

5.1 Contributions and Evaluation

The contributions made will be scoped with reference to the overarching purpose, as well as the objectives. Overall, the purpose of this thesis was to develop an interactive design tool for the implementation of cobot collaboration, that would be accessible to a non-technical audience. If the question were posed, “Was the tool developed?”, then the binary answer would be “yes”. Picking apart the purpose further, the tool is interactive, it does implement cobot collaboration. The accessibility to a non-technical audience depends wholly on if an audience which can obtain and use a Hololens is, by default, “technical” or not. One way to look at it is that, given the Hololens, a non-technical audience would be able to use the tool. Therefore, these smaller nuances in the purpose are deemed, on a very cursory and binary level, to be “done”.

However, it makes sense to rate progress more specifically by using the objectives set out after the purpose of the project. While the objectives themselves are not tied down to metrics, each objective may be given a qualitative score between 0% and 100% successful, with rationale provided for each. The three objectives were:

1. Present a physically and operationally accurate model of an available cobot.
2. Design a realistic manufacturing scenario which could be a good candidate for human-robot collaboration.
3. Provide an example interactive experience for a human with the cobot and manufacturing scenario

Objective 1 was attempted through Chapter 2, in development of the cobot and the accompanying Teach Pendant. At the most basic level, the cobot was the same size and shape as the real UR-10 by careful solid modelling techniques. The links were assembled in a kinematically-motivated serial chain through Hinge joints. The kinematics themselves were not quite up-to-par with a closed-form analytical solution, although being able to obtain usable inverse kinematics meant that the robot would be more than just a random assemblage of solid bodies. At times with a robot manipulator, it is desired to obtain different joint angles for adapting to the work environment. Unfortunately, while the Tool Flange inverse kinematics were modelled, the forward kinematics were not. This limited the UR-10's ability in some cases, as a poor assemblage of joints may have resulted in the rest of the robot arm colliding with the environment.

Collisions were modelled for the UR-10, although these collisions did not allow for a robust reaction from the cobot. If the cobot hit something, it would still "explode" even though it would eventually re-assemble itself. The only collisions that were appropriately modelled were on the Tool Flange, and only to non-anchored bodies like the door handle where the Fixed Joint was able to attach and detach. More broadly, the "dynamic" or kinetic behavior of the cobot was not modelled. The cobot was assumed to be a purely kinematic model, which while useful for prototyping, loses out on unique opportunities to test the strength and speed of the robot as it would behave in real life. Commentary about the Teach Pendant will be put on hold until rating the interactivity in Objective 3. As a final point, an end effector like a gripper was never added to the Tool Flange of the robot. Interacting with this would have been much better than "magnetically" attaching the handle to the Tool Flange.

The subjective ranking of the design tool rests with the balance between pure realism and functional realism. In terms of pure realism, Objective 1 would only be rated at about 30%. There is still a long way to go with modelling to truly say that the robot is physically accurate. However, from a functional standpoint, the robot does a lot of what one would need it to do to be at least somewhat useful for proof-of-concept. It seems fairer to grade based on functionality, rather than pure realism, as it is just a design tool. Therefore, Objective 1 is ranked 70% successful.

Objective 2 rests in the design of the manufacturing scenario. This can be split up into the idea itself, and then the implementation of such an idea. Significant forethought was given into the ideation of the scenario, with a defense of the concept in Chapter 3 Section 1. Among those

operations which are currently performed by collaborative robots, the idea of collaborating on finishing operations is perhaps less popular (in contrast to machine tending, pick-and-place, etc).

There may be some potential downsides to collaboration in a grinding and polishing scenario. First, it may be that the purchase of a robot for such a task may not be required, if the start-to-finish process bottleneck of making a door handle is not finishing. In example, if it takes 20 minutes to machine the handle, then it may be trivial to take less than a minute to finish it, and therefore does not have a very good return-on-investment overall. Second, improper grinding setup or programming with the cobot may damage it and its end effector. Just the same, exposing it to harsh vibrations and sparks may also fatigue the components and reduce its lifespan. In addition, there is nothing to say that if the finishing operation is enough of a process bottleneck, that two conventional or collaborative robots may be purchased which remove the need of the manufacturing associate to polish.

The implementation of the manufacturing scenario was parts successful and unsuccessful. Being able to rotate through the task with the cobot and human operator was a highly valued contribution toward the scenario. Physical modelling of the bench grinder received a lot of attention and appears very polished and fully-functional. However, the handle modelling itself was lackluster. The deformation of grinding was an illusion at best, and the polishing transformation was oversimplified. The robot interacting with the environment was also cumbersome, compounded with the inability to “grasp” a part but instead “magnetize” it. For all these reasons, the manufacturing scenario may be ranked 65% successful. Holding the scenario constant and polishing up the implementation, it may rise to 90% successful.

The last objective, Objective 3, was the interactivity of the design tool. This is a combination of both core developments in the cobot and Teach Pendant, and the manufacturing scenario. Overall, there is enough interactivity with the user for them to be able to accomplish the scenario at hand, through teaching the robot and playing their own role as manufacturing associate. However, a shortcoming is that the design tool is not built with any sort of guidance as to what the user should be doing. For the user to interact well with the tool and its components, some level of training, instruction, or trial-and-error is necessary. Another shortcoming is that the tool can be unforgiving when it comes to user error. If the user crashes the robot into the bench grinder when it’s running, for instance, it’s very likely that the robot will become stuck inside. Or, if the user is trying to teach the robot, and they mess up an action, there is no way to go back and change it without re-recording the entire routine. To balance out these two perspectives, a score of 75% is given to Objective 3.

5.2 Summary

A new horizon in manufacturing, Industry 4.0, has big ideas for the future of manufacturing and its implications on society [2]. Part of that vision is the shared labor of man and machine to best accomplish tasks. There is a general lack of understanding as to how this would be done, especially by small-to-medium sized enterprises, who have not traditionally used robotics due to barriers in safety and programming. The thesis aimed to create a design tool for better understanding implementation of cobots to non-technical users. The candidate technology for this tool was virtual and augmented reality, providing a more immersive understanding and opening up access to those

without a technical background. This was done by first modelling and implementing the cobot, a UR-10 collaborative robot manipulator, along with its corresponding Teach Pendant. Then, a candidate manufacturing scenario of a door handle finishing operation was modelled and implemented. The tool was tested on the Hololens in a demonstration to confirm its success. In short, the design tool was developed for this purpose. Based on its objective rankings, it does a fair job, but there are many avenues for improvement.

5.3 Considerations for Future Work

If the same tool is to be used, any number of changes could be done to help improve it. Many of these changes are listed in Section 1 of this Chapter, and are sorted by objective so that an interested person can decide which aspects of the tool to improve. Using this thesis as a reference, along with access to files used to work up until this point, one should be able to continue to add new features. However, they may find themselves limited by the capabilities of Unity without diving into the details of its backend. For instance, modelling the kinetics of the UR-10 may require intimate knowledge into the physics engine used, PhysX.

The methods used for implementation of modelled elements seen in the thesis, while perhaps unique, do not bind the implementation to these methods. The major challenge in the field of VR/AR modelling for the purposes of architecture, science, engineering, and business is the ability to convert existing and well-defined solutions into solutions for VR/AR.

For example, the hinge method used in Unity, while crude, effective and does not require analytical kinematics. While analytical kinematics are indeed useful, they are often difficult to derive from first principles engineering, and perhaps even more difficult to implement. Tools like SOLIDWORKS and MathWorks' Simscape Multibody provide a way to automatically convert hinge joints, and other geometric constraints, into code. Being able to fully automate the conversion process between solid model and working behavioral multibody physics in VR/AR would greatly ease this process. Integrating more functionality nearly necessitates making other software packages collaborate, in some form, with VR/AR platforms. This would greatly accelerate the pace of design and remove the required technical expertise from the equation.

Finally, it should not be understated that work in VR/AR should be done not on an individual basis, but on a diverse and multidisciplinary team. This is how robotics has advanced to the abilities it now has today, when varied technical backgrounds come together to solve a problem. Artists, designers, psychologists and teachers, to name several, can also contribute their abilities in helping us all understand what possibilities virtual and augmented reality hold.

Appendix A: Introduction to Unity

Unity is a real-time engine used to design video games and related digital content. The version used for this thesis was Unity 2017.3.1f1 Personal, which is freely available through the Unity website, https://store.unity.com/download. Upon loading a new or existing solution, the Unity Editor will appear for the project. Figure 48 below explains some of the key windows in the Unity Editor layout.

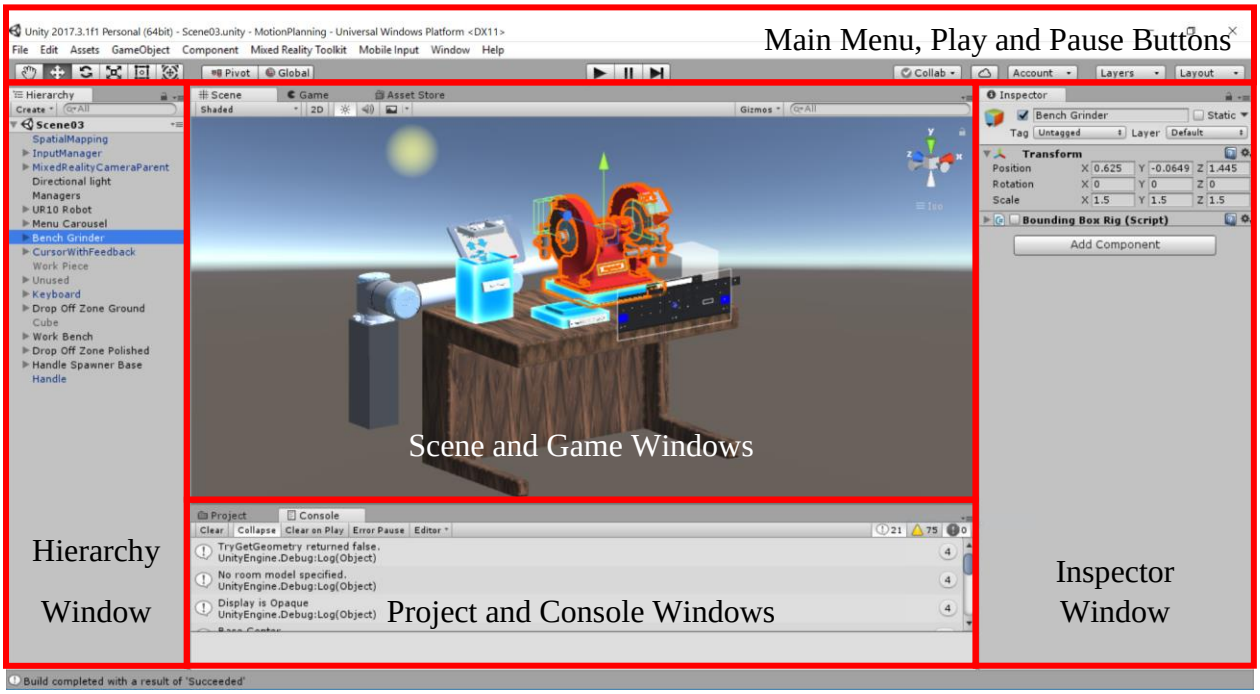


Figure 48 Main screen of the Unity Editor, with key windows.

Digital elements that are available in Unity are known as Assets. All the Assets can be viewed through the Project window, which organizes them by folders. In the root folder of this project, for example, are some of the solid body objects and scripts that were created for the thesis. Many Assets are provided to the user automatically when the start a project.

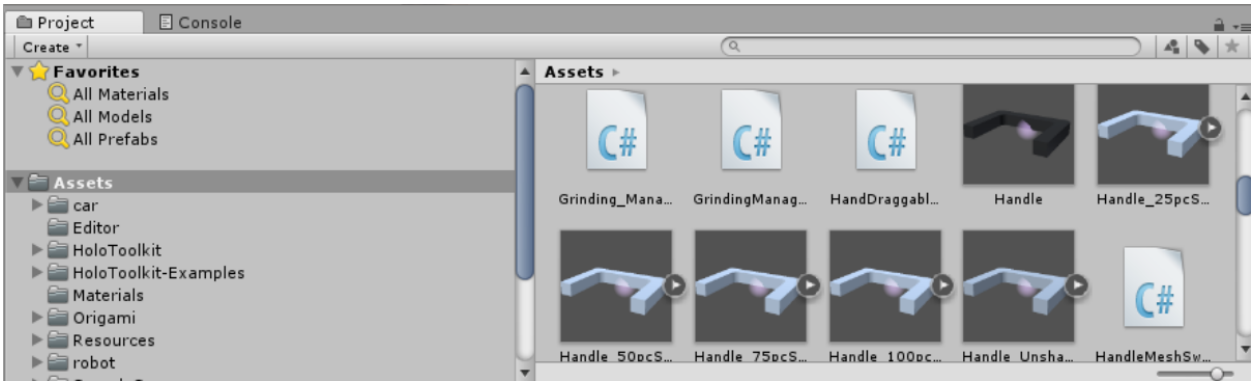


Figure 49 Project window with Assets

A bundled collection of Assets, known as a Package, can be imported into Unity, and is often given its own folder. Many Packages exist that aid in the acceleration of software design. In this project for the thesis, three Packages were downloaded:

- Unity's own Standard Assets Package, which provides many off-the-shelf solutions available for game design. This was lesser used. One can access it directly from Unity, under the Assets tab.
- The Microsoft Mixed Reality Toolkit (MRTK) Package for Unity, Holotoolkit, which support essential Assets for VR/AR design in Unity. https://github.com/Microsoft/MixedRealityToolkit-Unity
- A Package filled with examples for the Holotookit Package, Holotoolkit-Examples. https://github.com/Microsoft/MixedRealityToolkit-Unity/tree/master/Assets/HoloToolkit-Examples

These three gave a great start in bringing in useful, "pre-fabricated" Assets, or Prefabs.

However, some Assets either must be digitally imported, or created directly in Unity. Even so, some Assets that have been imported should be modified. This is where the Hierarchy and Inspector come in. The object Heirarchy, seen through the Hierarchy window, contains all the objects in the Scene. For ease, assume that the project only has one Scene or "world".

Figure 50 A portion of the Hierarchy window

Anything that is listed directly in the Hierarchy is defined as a GameObject. A GameObject can be either a singular object, such as the Directional Light, or a "parent" GameObject made *with* one or more GameObject "child(ren)". A child GameObject can also be a parent of another GameObject, and so on, until there are only children GameObjects at the bottom.

Clicking on a GameObject will bring up information about it in the Inspector window. The Inspector can toggle whether the GameObject is active or not in the scene. More importantly, the Inspector shows all of the properties of the GameObject, which are called Components. Every

GameObject comes attached with a Transform Component, which defines its location within the world. For example, the UR10 Robot GameObject below only contains the Transform Component. This type of GameObject is often called an “Empty” GameObject and is used mostly to group things together. In this case, the UR10 Robot GameObject contains all the links of the robot.

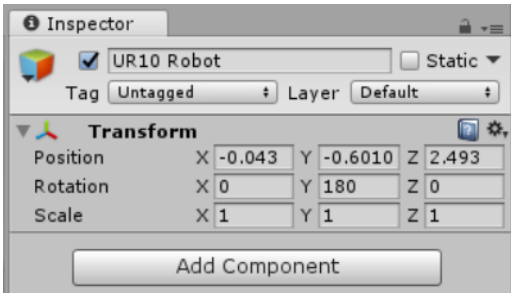
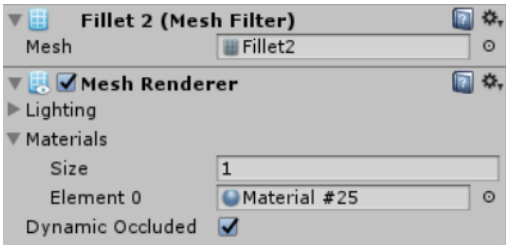


Figure 51 Inspector window when the UR10 Robot GameObject is selected

Some very common Components are used within this thesis, and so the user should familiarize themselves with them and their functionality. The vast majority of Components, if either included with Unity or obtained through a Package, may be researched online by name for reference.

Take, for example, the Handle GameObject in this thesis. It was modelled in SOLIDWORKS and is brought into Unity. For any 3D objects, two components are needed to ensure that it is visualized correctly: the MeshFilter and the MeshRender Components.



The MeshFilter accepts a Mesh parameter, which is either automatically generated (in example, a Cube is pretty easy) or imported for 3D models. In the example of the Handle, the Mesh from was named after the last feature logged, which was Fillet2.

Figure 52 (Left) Visualization of Handle in Unity Scene, (Right) Imported Fillet2 Mesh

Meanwhile, the MeshRender assigns the Material parameter to overlay the MeshFilter. So, if Material #25 is basically, “metal gray”, then that is what the Handle will be colored like.

If only left with the Transform, MeshFilter and MeshRenderer, then there will be a floating 3D model of the Handle in the Scene. This is not very realistic, so we desire to model the handle using rigid body mechanics. Unity has a Rigidbody Component just for this purpose. With it, some basic properties of the body are presented, like Mass, Drag, and Angular Drag. The object may also Use Gravity, to make the object appear weighted. The Rigidbody can also be set to “Is Kinematic”, which prevents it from acting under the influence of all forces.

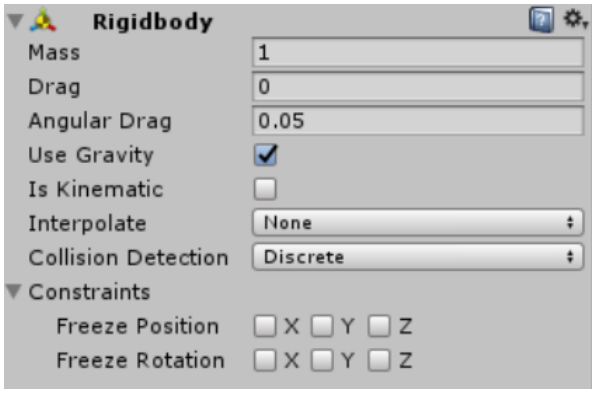


Figure 53 Rigidbody Component with default parameter settings

Assuming the Handle has the Rididbody attached, it will fall but will not interact with the environment. This is because the Handle was not given any collision information. One could assume that the Mesh would be used by default for the handle – however, calculating collisions with a high polygon Mesh is often computationally burdensome. A variety of Collision options are available and are presented in Chapter 2.1. For the moment, the three pieces of the Handle can be modelled by incorporating three Box Collider Components. They can be scaled and placed with respect to the GameObject Transform. It should be stated that GameObjects do not need to have a Rigidbody component to collide with one another, although no physical meaning will be associated to the collision (i.e. transfer of momentum) without a Rigidbody component attached.

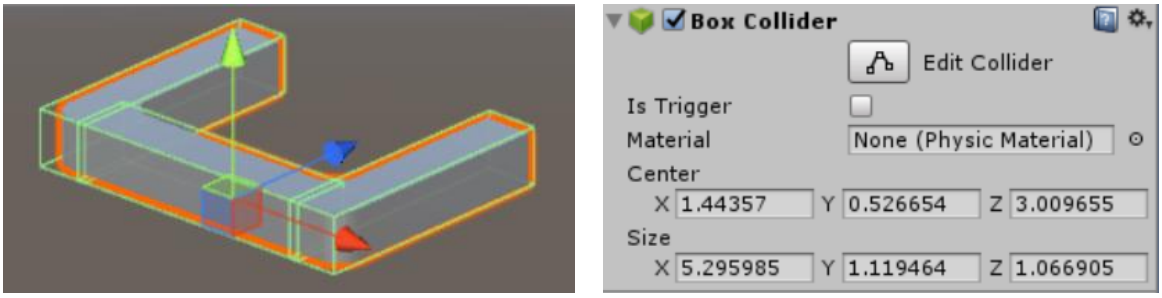


Figure 54 (Left) Handle surfaces defined by three Box Colliders, (Right) One of the Box Colliders of the Handle

GameObjects with Rigidbody Components can interact in more interesting ways than just collisions. Various kinds of Joints are available in Unity, like Slider or Hinge joints. They will be discussed as needed in the thesis.

The last component to learn about is a C# Script. The Unity Editor is built upon C# and so are all the Components. C# is an object-oriented programming language, which makes creating scripts for Unity easy. C# scripts can accept almost any parameter, including other GameObjects. This gives them the most flexibility of any Component.

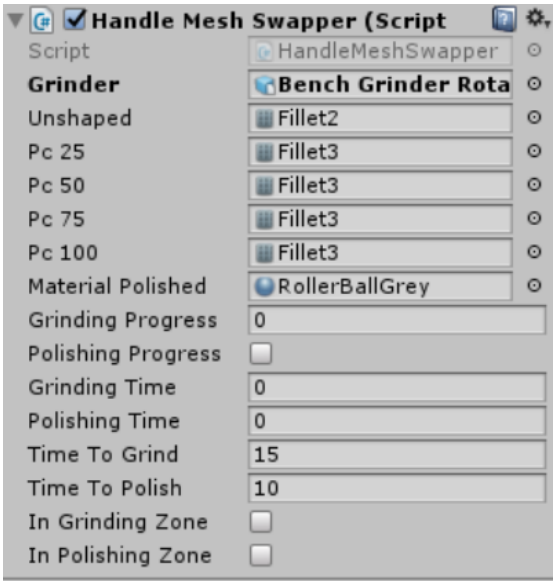


Figure 55 Example of a C# Script Component named HandleMeshSwapper, attached to the Handle GameObject

In addition, every time a new C# Script is created in Unity, all classes have access to Unity's built-in functions through the MonoBehavior namespace. Code which is meant to run continuously occurs in the Update or FixedUpdate loop, while code meant to run only once typically occurs in the Start loop. C# code is edited through Microsoft Visual Studio, which is required for Unity.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class NewScript : MonoBehaviour {

	// Use this for initialization
	void Start () {

	}

	// Update is called once per frame
	void Update () {

	}
}
```

Figure 56 New script component in Unity, opened with Visual Studio

Appendix B: Solid Model Export to Unity Process

The primary solid model design tool used in this thesis is SOLIDWORKS (specifically, SOLIDWORKS Premium 2017 x64 Edition). A process for exporting SOLIDWORKS part files to Unity was developed during this thesis and was also used in [13]. Autodesk 3ds Max was used to perform the conversion process. The process will read as a set of instructions.

To begin, it is necessary to note the orientation and base units of the part in SOLIDWORKS. Find the direction which is "up" for the part in question. If the part was modelled on the XZ-plane of SOLIDWORKS per convention, then the Y-axis would be "up".

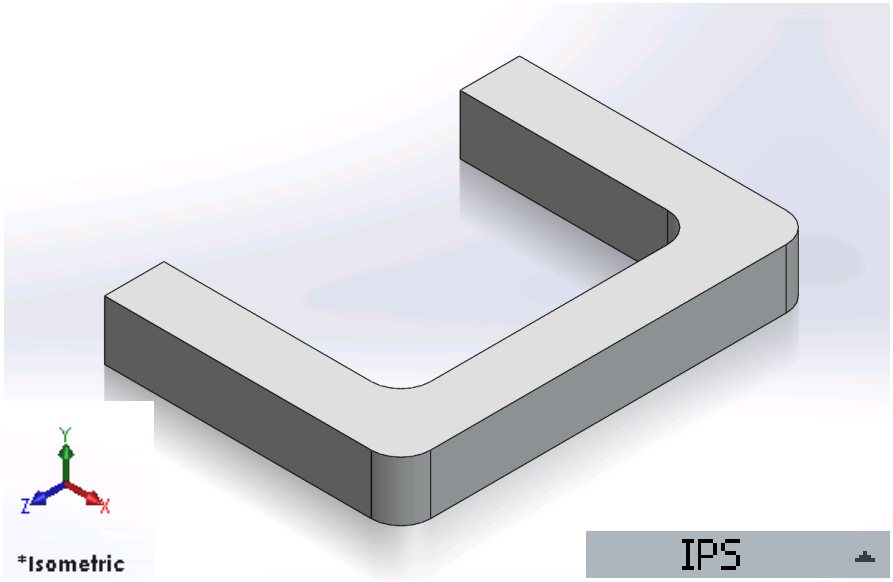


Figure 57 Handle SOLIDWORKS part, with coordinate system and unit system magnified

There are some things that one should be aware of before continuing the conversion process:

- Perform any colorations on the parts in SOLIDWORKS if that is easiest. Textures and finishes will not be imported into 3ds Max, only the colors will stay.
- The origin of the part in SOLIDWORKS will be the origin upon import into Unity and cannot be changed. Make changed to the model if that is a problem. In example, parts which rotate about any particular axis will want the part origin to intersect the axis of rotation.
- The SOLIDWORKS coordinate system obeys the right-hand rule, while the Unity coordinate system obeys the left-hand rule. Functionally, this means that the X and Z-axes are flipped when comparing SOLIDWORKS and Unity, which can be corrected for later.

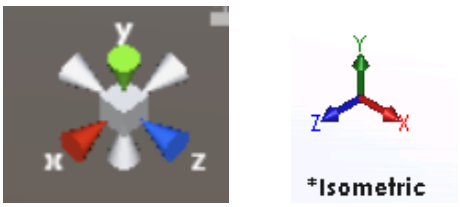


Figure 58 Coordinate system in Unity versus SOLIDWORKS

Open 3ds Max. Assure that the file format for the SOLIDWORKS part is *.sldprt for best results. To import, on the main menu ribbon on the top-left corner, go under “File”, “Import”, “Import…” and a Windows Explorer box will appear. Select the *.sldprt file and click “Open”. Upon doing so, an Import Settings box will appear. While there are no hard rules about the import settings, the following are suggested.

- Set “Convert to Mesh” as “On”
- Set “Up Axis” to the corresponding “up” axis in SOLIDWORKS (normally, Y-Up)
- Set “Hierarchy Mode” to “Flattened” for all the part geometries to be on the same object level. This is recommended to keep things easy, although other options can be explored
- Set “Mesh Resolution” to -10, to reduce the polygon count upon import
- Set “Keep Dummy Nodes” as checked

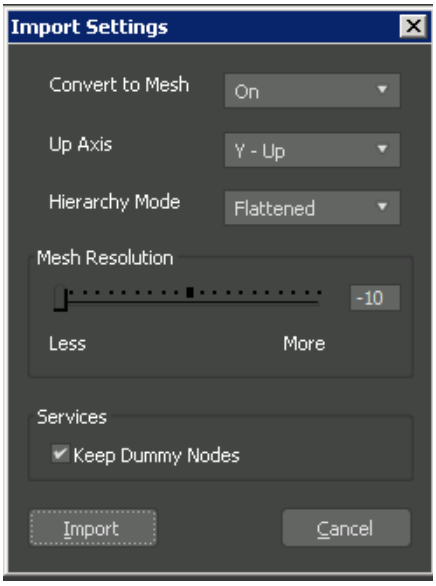


Figure 59 SOLIDWORKS part import settings into 3ds Max

The imported part can be viewed through the Perspective viewport in 3ds Max. Notice that in this coordinate system, the up-axis is Z, not Y. The colors are often close to what the color will be in Unity, so changes in either SOLIDWORKS or 3ds Max can be made prior to export.

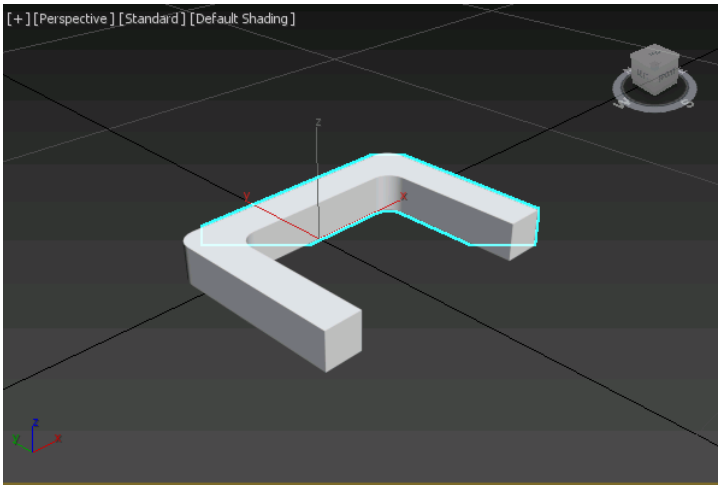


Figure 60 View of imported part in 3ds Max

If all looks good, then it is time to export. Under “File”, “Export”, “Export…”, another Windows Explorer box will appear as a “Save As…” menu. Name the file and under “Save as type”, select “Autodesk (*.FBX)”. The FBX format is especially helpful for video games, as it exports the mesh and material files for all of the polygon models. Clicking “Save” will make a box appear with “FBX Export”. Most of the settings are unimportant, except for “Units” and “Axis Conversion”, which can be found under “Advanced Settings”.

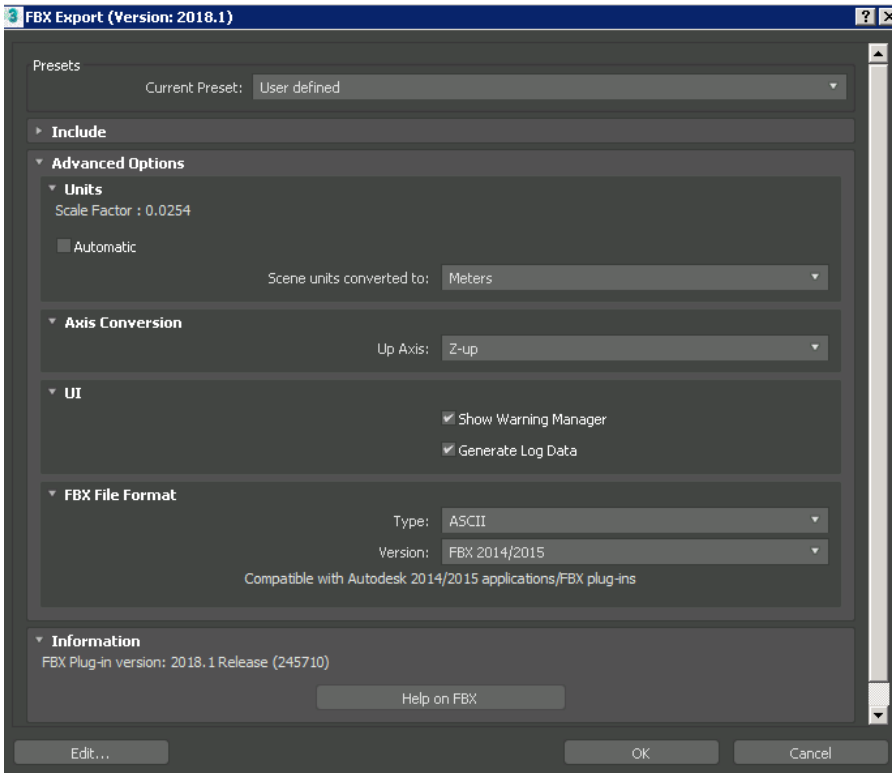


Figure 61 FBX Export options

The base units in Unity are in meters, and the coordinate system is Y-up, just like in SOLIDWORKS. The unit conversion process in this model correctly assumed that the Handle model had units in inches, although this is not always going to be the case. Ensure that the proper scale factor is given such that the scale factor times the fundamental unit of the model (in, mm, cm, m) will result in an new unit of 1 m. For the coordinate system, the up axis in the model should be Z-up, assuming the part was imported correctly into 3ds Max. If it was not, then this may be an opportunity to correct that error.

When finished, click "OK". This will prompt 3ds Max to render the FBX file. When it is complete, the file is ready to import. In Unity, simply go to the main menu ribbon on the top, select "Assets", "Import New Asset…", and click "Open" after selecting the FBX file. A new Prefab object will be loaded into the Project folder, and additional settings will be presented in Unity, although these do not need to be changed usually. That's it!

Appendix C: Hololens Gestures

The Hololens can detect “Gestures”, which are specific motions of the user’s hand or hands [18]. Using Gestures, the Hololens and apps which are running on it can be controlled. Gestures are captured by the cameras on the Hololens, and so the user’s hand(s) need to be in front of them. The two core Gestures that the Hololens uses are the “Air Tap” and “Bloom”.

The Bloom Gesture is performed by first holding one’s hand “palm up, with [their] fingertips together, then open[ing] [their] hand” [18]. The figure displays both states needed to Bloom. Bloom is reserved for the Hololens as a menu option, so it has no function in any app.

Figure 62 Hololens Bloom Gesture [18]

The Air Tap Gesture is performed through two states – the ready state, which is made by lifting up one’s index finger, and the pressed state, which is made by pressing down the finger.

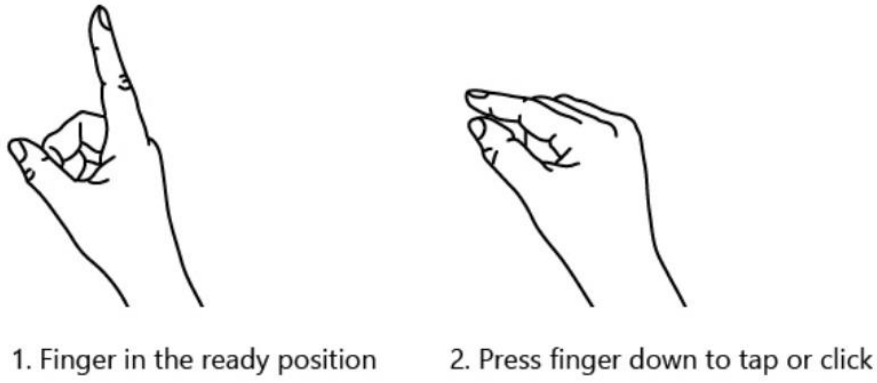


Figure 63 Hololens Air Tap Gesture with steps listed [18]

An Air Tap functions much like the left mouse click on a computer. Done quickly, the action will be performed once. A Composite Gesture, “Tap and Hold”, is made when the pressed state is not released for some time – like holding down the mouse click. The analog to the “pointer” of the mouse is instead the user’s Gaze, which is often a cursor projected from their center of view onto holograms in the environment. Unity is able to receive Gesture information from the Hololens, which helps in developing VR/AR apps that the user can interact with.

References

[1] University of California San Diego, Contextual Robotics Institute, "A Roadmap for US Robotics, From Internet to Robotics," 2016.

[2] S. Jeschke, "Robotics in Industry 4.0: History, Presence and Future of Robotics in Car Industry," RWTH Aachen University, Lüneberg, 2016.

[3] A. B. Moniz and B.-J. Krigs, "Robots Working with Humans or Humans Working with Robots? Searching for Social Dimensions in New Human-Robot Interaction in Industry," *Societies,* vol. 23, no. 6, p. 21, 2016.

[4] Inverse, "The Tesla Factory Is a Symphony of Hands-on Engineers and Superhero Robots," 15 February 2018. [Online]. Available: https://www.inverse.com/article/41363-tesla-factory-robots-engineers.

[5] S. J., Getty Images, 2015.

[6] Business Wire, "New DENSO Robotics WINCAPS III How-To Videos," 22 October 2013. [Online]. Available: https://www.businesswire.com/news/home/20131022005546/en/New-DENSO-Robotics-WINCAPS-III-How-To-Videos.

[7] Robotiq, "What is an Average Price for a Collaborative Robot ?," 3 February 2016. [Online]. Available: https://blog.robotiq.com/what-is-the-price-of-collaborative-robots.

[8] New Atlas, "“Baxter is a relatively inexpensive new industrial robot, that can reportedly be trained to perform tasks by regular people.” .".

[9] Brink News, "Risk Assessment for ‘Safe’ Collaborative Robots Still Needed," 4 September 2015. [Online]. Available: https://www.brinknews.com/risk-assessment-for-safe-collaborative-robots-still-needed/.

[10] John Hopkins University Laboratory for Computational Sensing and Robotics, "Programming an Industrial Robot using the Oculus Rift," [YouTube Video], 6 March 2014. [Online]. Available: https://www.youtube.com/watch?v=uyGJ7gd7jq8. [Accessed 23 September 2017].

[11] ProFactor, "KoMoProd Human Robot Collaboration - Profactor GmbH," [YouTube video], 6 June 2016. [Online]. Available: https://www.youtube.com/watch?v=URfJUMNc9SY. [Accessed 9 September 2017].

[12] Engadget, "Microsoft's mixed reality is for developers, not the public," 17 December 2015. [Online]. Available: https://www.brinknews.com/risk-assessment-for-safe-collaborative-robots-still-needed/.

[13] C. Spicer, "Development of an Augmented Reality Testing Platform for Collaborative Robots," 2018.

[14] Universal Robots, "Technical specifications UR10," October 2014. [Online]. Available: https://www.universal-robots.com/media/50880/ur10_bz.pdf.

[15] K. Hawkins, "Analytic Inverse Kinematics for the Universal Robots," 7 December 2013. [Online]. Available: https://smartech.gatech.edu/bitstream/handle/1853/50782/ur_kin_tech_report_1.pdf.

[16] Cross, "Use a Robot....but Why?," 29 April 2015. [Online]. Available: https://www.crossco.com/blog/use-robotbut-why.

[17] dea (GrabCad user), "GrabCad," 30 January 2016. [Online]. Available: https://grabcad.com/library/bench-grinder-6.

[18] Microsoft Windows Dev Center, "Gestures," 20 March 2018. [Online]. Available: https://docs.microsoft.com/en-us/windows/mixed-reality/gestures.

[19] UK-RAS Network, "Manufacturing Robotics: The Next Robotic Industrial Revolution," 2016.

[20] Universal Robots A/S, "UR10 robot: A Collaborative Industrial Robot," 2017. [Online]. Available: https://www.universal-robots.com/products/ur10-robot/. [Accessed 24 September 2017].

[21] Microsoft, "Microsoft Hololens," 2017. [Online]. Available: https://www.microsoft.com/en-us/hololens/hardware. [Accessed 25 September 2017].